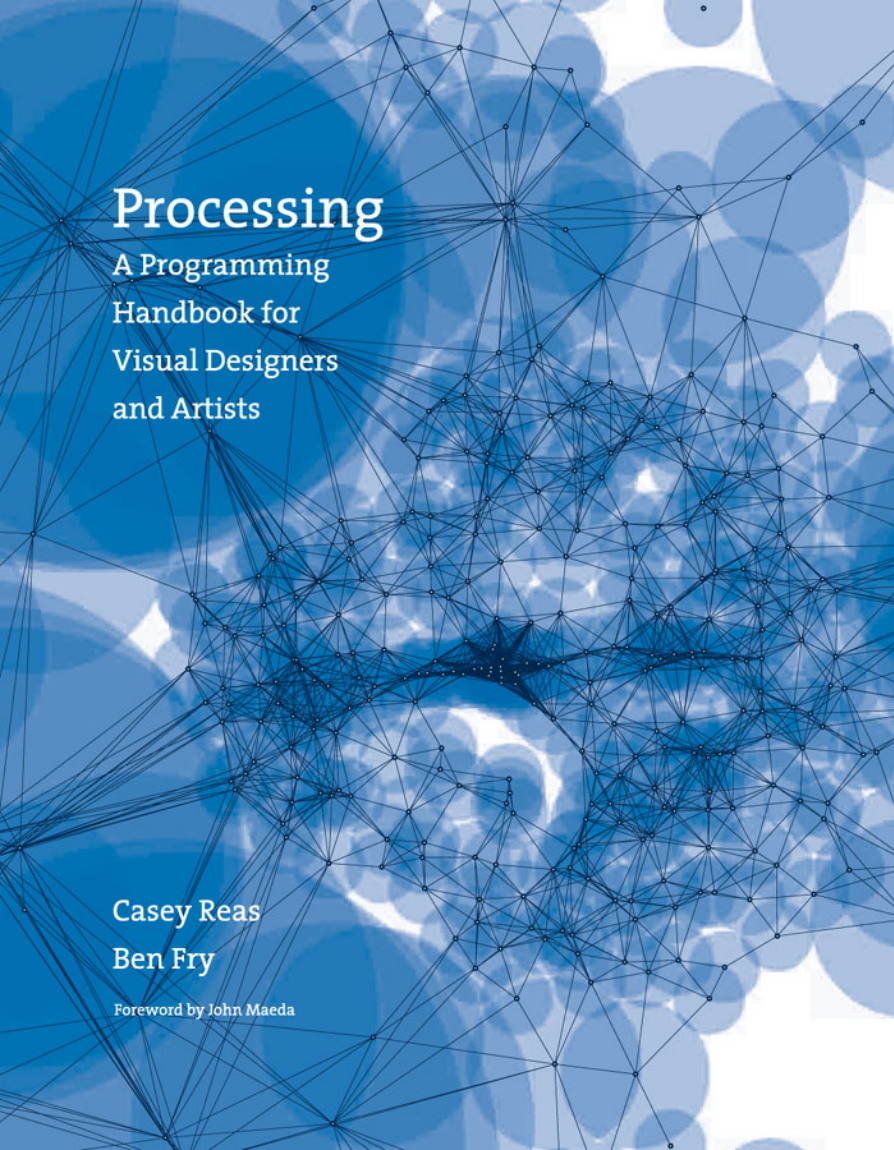


Processing

A Programming Handbook for Visual Designers and Artists

Casey Reas
Ben Fry

Foreword by John Maeda

Processing: a programming handbook for visual designers and artists

Casey Reas
Ben Fry

The MIT Press
Cambridge, Massachusetts
London, England

© 2007 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

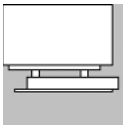
Reas, Casey.
Processing : a programming handbook for visual designers and artists / Casey Reas & Ben Fry ; foreword by John Maeda.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-18262-1 (hardcover : alk. paper)
1. Computer programming. 2. Computer graphics—Computer programs. 3. Digital art—Computer programs. 4. Art—Data processing. 5. Art and technology. I. Fry, Ben. II. Title.

QA76.6.R4138 2007
005.1—dc22

2006034768

10 9 8 7 6 5 4 3 2 1

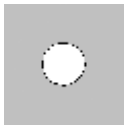
For the ACG



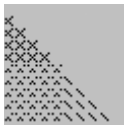
29

34

45

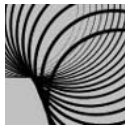


57



67

72



84

91

99

113

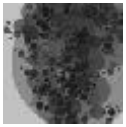
121

131

141

189

192



204

208

221

225

233

244

247

289

297

307

320

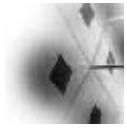
324

331

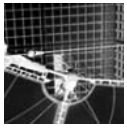
336

344

352



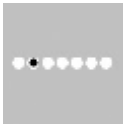
354



359

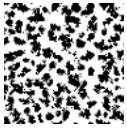
409

415



447

451



472

493

530

535

551

Contents

xix Foreword
xxi Preface

1 Processing…
9 Using Processing

17 Structure 1: Code Elements
23 Shape 1: Coordinates, Primitives
37 Data 1: Variables
43 Math 1: Arithmetic, Functions
51 Control 1: Decisions
61 Control 2: Repetition
69 Shape 2: Vertices
79 Math 2: Curves
85 Color 1: Color by Numbers
95 Image 1: Display, Tint
101 Data 2: Text
105 Data 3: Conversion, Objects
111 Typography 1: Display
117 Math 3: Trigonometry
127 Math 4: Random
133 Transform 1: Translate, Matrices
137 Transform 2: Rotate, Scale
145 Development 1: Sketching, Techniques
149 Synthesis 1: Form and Code
155 Interviews 1: Print

173 Structure 2: Continuous
181 Structure 3: Functions
197 Shape 3: Parameters, Recursion
205 Input 1: Mouse I
217 Drawing 1: Static Forms
223 Input 2: Keyboard
229 Input 3: Events
237 Input 4: Mouse II
245 Input 5: Time, Date
251 Development 2: Iteration, Debugging
255 Synthesis 2: Input and Response
261 Interviews 2: Software, Web

279 Motion 1: Lines, Curves
291 Motion 2: Machine, Organism
301 Data 4: Arrays
315 Image 2: Animation
321 Image 3: Pixels
327 Typography 2: Motion
333 Typography 3: Response
337 Color 2: Components
347 Image 4: Filter, Blend, Copy, Mask
355 Image 5: Image Processing
367 Output 1: Images
371 Synthesis 3: Motion and Arrays
377 Interviews 3: Animation, Video

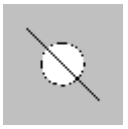
395 Structure 4: Objects I
413 Drawing 2: Kinetic Forms
421 Output 2: File Export
427 Input 6: File Import
435 Input 7: Interface
453 Structure 5: Objects II
461 Simulate 1: Biology
477 Simulate 2: Physics
495 Synthesis 4: Structure, Interface
501 Interviews 4: Performance, Installation

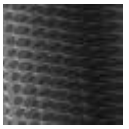
519 Extension 1: Continuing…
525 Extension 2: 3D
547 Extension 3: Vision
563 Extension 4: Network
579 Extension 5: Sound
603 Extension 6: Print
617 Extension 7: Mobile
633 Extension 8: Electronics

661 Appendixes
693 Related Media
699 Glossary
703 Code Index
705 Index

88

342

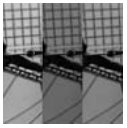

55


65

305

220

415


98

319

323

351

353

359

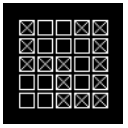
207

225

232

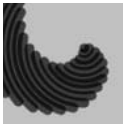
240

247


444

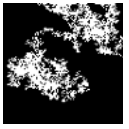
44

83


124

129

288


296

29

32

75

202

470

488

184

190

407

455

141

113

329

335

530

535

551

Contents by category

xix Foreword
xxi Preface

1 Processing…
9 Using Processing

85 Color 1: Color by Numbers
337 Color 2: Components
51 Control 1: Decisions
61 Control 2: Repetition
37 Data 1: Variables
101 Data 2: Text
105 Data 3: Conversion, Objects
301 Data 4: Arrays
145 Development 1: Sketching, Techniques
251 Development 2: Iteration, Debugging
217 Drawing 1: Static Forms
413 Drawing 2: Kinetic Forms
95 Image 1: Display, Tint
315 Image 2: Animation
321 Image 3: Pixels
347 Image 4: Filter, Blend, Copy, Mask
355 Image 5: Image Processing
205 Input 1: Mouse I
223 Input 2: Keyboard
229 Input 3: Events
237 Input 4: Mouse II
245 Input 5: Time, Date
427 Input 6: File Import
435 Input 7: Interface
43 Math 1: Arithmetic, Functions
79 Math 2: Curves
117 Math 3: Trigonometry
127 Math 4: Random
279 Motion 1: Lines, Curves
291 Motion 2: Machine, Organism
367 Output 1: Images
421 Output 2: File Export

23 Shape 1: Coordinates, Primitives
69 Shape 2: Vertices
197 Shape 3: Parameters, Recursion
461 Simulate 1: Biology
477 Simulate 2: Physics
17 Structure 1: Code Elements
173 Structure 2: Continuous
181 Structure 3: Functions
395 Structure 4: Objects I
453 Structure 5: Objects II
149 Synthesis 1: Form and Code
255 Synthesis 2: Input and Response
371 Synthesis 3: Motion and Arrays
495 Synthesis 4: Structure, Interface
133 Transform 1: Translate, Matrices
137 Transform 2: Rotate, Scale
111 Typography 1: Display
327 Typography 2: Motion
333 Typography 3: Response

155 Interviews 1: Print
261 Interviews 2: Software, Web
377 Interviews 3: Animation, Video
501 Interviews 4: Performance, Installation

519 Extension 1: Continuing…
525 Extension 2: 3D
547 Extension 3: Vision
563 Extension 4: Network
579 Extension 5: Sound
603 Extension 6: Print
617 Extension 7: Mobile
633 Extension 8: Electronics

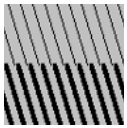
661 Appendixes
693 Related Media
699 Glossary
703 Code Index
705 Index

29

30

44

55



63

70

83

88

97

113

124

128

137

174

186

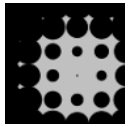
200

206

219

225

231



239

246

281

293

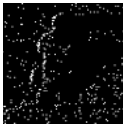
306

316

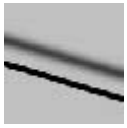
322

329

334



340

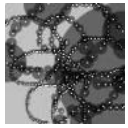


349

353

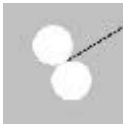
356

406

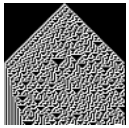


414

441



458



464

484

530

535

551

Extended contents

xix Foreword by John Maeda

xxi Preface
xxi Contents
xxii How to read this book
xxiii Casey's introduction
xxiv Ben's introduction
xxv Acknowledgments

1 Processing…
1 Software
3 Literacy
4 Open
4 Education
6 Network
7 Context

9 Using Processing
9 Download, Install
9 Environment
10 Export
11 Example walk-through
16 Reference

17 Structure 1: Code Elements
17 Comments
`//, /* */`
18 Functions
18 Expressions, Statements
`";", ","`
20 Case sensitivity
20 Whitespace
20 Console
`print(), println()`

23 Shape 1: Coordinates, Primitives
23 Coordinates
`size()`
25 Primitive shapes
`point(), line(), triangle(), quad(), rect(), ellipse(), bezier()`
31 Drawing order
31 Gray values
`background(), fill(),stroke(), noFill(), noStroke()`
33 Drawing attributes
`smooth(), noSmooth(), strokeWeight(), strokeCap(), strokeJoin()`
34 Drawing modes
`ellipseMode(), rectMode()`

37 Data 1: Variables
37 Data types
`int, float, boolean, true, false`
38 Variables
`=`
40 Processing variables
`width, height`

43 Math 1: Arithmetic, Functions
43 Arithmetic
`+, -, *, /, %`
47 Operator precedence, Grouping
`()`
48 Shortcuts
`++, --, +=, -=, *=, /=, -`
49 Constraining numbers
`ceil(), floor(), round(), min(), max()`

51 Control 1: Decisions
51 Relational expressions
`>, <, >=, <=, ==, !=`
53 Conditionals
`if, else, {}`
57 Logical operators
`||, &&, !`

61 Control 2: Repetition
61 Iteration
`for`
65 Nested iteration
67 Formatting code blocks

69 Shape 2: Vertices
69 Vertex
`beginShape(), endShape(), vertex()`
71 Points, Lines
72 Shapes
74 Curves
`curveVertex(), bezierVertex()`

79 Math 2: Curves
79 Exponents, Roots
`sq(), sqrt(), pow()`
80 Normalizing, Mapping
`norm(), lerp(), map()`
83 Simple curves

85 Color 1: Color by Numbers
86 Setting colors
89 Color data
`color, color()`
89 RGB, HSB
`colorMode()`
93 Hexadecimal

95 Image 1: Display, Tint
96 Display
`PImage, loadImage(), image()`
97 Image color, Transparency
`tint(), noTint()`

101 Data 2: Text
102 Characters
`char`
103 Words, Sentences
`String`

105 Data 3: Conversion, Objects
105 Data conversion
`boolean(), byte(), char(), int(), float(), str()`
107 Objects
“.”,
`PImage.width, PImage.height, String.length, String.startsWith(), String.endsWith(); String.charAt(), String.toCharArray(), String.subString(), String.toLowerCase(), String.toUpperCase(), String.equals()`

111 Typography 1: Display
112 Loading fonts, Drawing text
`PFont, loadFont(), textFont(), text()`
114 Text attributes
`textSize(), textLeading(), textAlign(), textWidth()`

117 Math 3: Trigonometry
117 Angles, Waves
`PI, QUARTER_PI, HALF_PI, TWO_PI, sin(), cos(), radians(), degrees()`
123 Circles, Arcs, Spirals
`arc()`

127 Math 4: Random
127 Unexpected numbers
`random(), randomSeed()`
130 Noise
`noise(), noiseSeed()`

133 Transform 1: Translate, Matrices
133 Translation
translate()
134 Controlling transformations
pushMatrix(), popMatrix()

137 Transform 2: Rotate, Scale
137 Rotation, Scaling
rotate(), scale()
139 Combining transformations
142 New coordinates

145 Development 1: Sketching, Techniques
145 Sketching software
146 Programming techniques

149 Synthesis 1: Form and Code
150 Collage Engine
151 Riley Waves
152 Wilson Grids
153 Mandelbrot Set

155 Interviews 1: Print
157 Jared Tarbell.
Fractal.Invaders, *Substrate*
161 Martin Wattenberg.
Shape of Song
165 James Paterson.
The Objectivity Engine
169 LettError.
RandomFont Beowolf

173 Structure 2: Continuous
173 Continuous evaluation
draw(), frameRate(),
frameCount
177 Controlling the flow
setup(), noLoop(),
178 Variable scope

181 Structure 3: Functions
182 Abstraction
183 Creating functions
void
193 Function overloading
194 Calculating and returning values
return

197 Shape 3: Parameters, Recursion
197 Parameterized form
201 Recursion

205 Input 1: Mouse I
205 Mouse data
mouseX, mouseY,
pmouseX, pmouseY
212 Mouse buttons
mousePressed, mouseButton
213 Cursor icon
cursor(), noCursor()

217 Drawing 1: Static Forms
218 Simple tools
221 Drawing with images

223 Input 2: Keyboard
224 Keyboard data
keyPressed, key
227 Coded keys
keyCode

229 Input 3: Events
229 Mouse events
mousePressed(),
mouseReleased(),
mouseMoved(), mouseDragged()
232 Key events
keyPressed(), keyReleased()
235 Controlling the flow
loop(), redraw()

237 Input 4: Mouse II
237 Constrain
constrain()
238 Distance
dist()
239 Easing
abs()
242 Speed
243 Orientation
atan2()

245 Input 5: Time, Date
245 Seconds, Minutes, Hours
second(), minute(), hour(), millis()
249 Date
day(), month(), year()

251 Development 2: Iteration, Debugging
251 Iteration
252 Debugging

255 Synthesis 2: Input and Response
256 Tennis
257 Cursor. Peter Cho
258 Typing
259 Banded Clock. Golan Levin

261 Interviews 2: Software, Web
263 Ed Burton. *Sodaconstructor*
267 Josh On. *They Rule*
271 Jürg Lehni. *Hektor and Scriptographer*
275 Auriea Harvey and Michaël Samyn. *The Endless Forest*

279 Motion 1: Lines, Curves
279 Controlling motion
284 Moving along curves
287 Motion through transformation

291 Motion 2: Machine, Organism
291 Mechanical motion
295 Organic motion

301 Data 4: Arrays
303 Using arrays
Array, [], new, Array.length
306 Storing mouse data
309 Array functions
append(), shorten(), expand(), arraycopy()
312 Two-dimensional arrays

315 Image 2: Animation
316 Sequential images
319 Images in motion

321 Image 3: Pixels
321 Reading pixels
get()
324 Writing pixels
set()

327 Typography 2: Motion
327 Words in motion
331 Letters in motion

333 Typography 3: Response
333 Responsive words
335 Responsive letters

337 Color 2: Components
337 Extracting color
red(), blue(), green(), alpha(), hue(), saturation(), brightness(),
341 Dynamic color palettes

347 Image 4: Filter, Blend, Copy, Mask
347 Filtering, Blending
filter(), blend(), blendColor()
353 Copying pixels
copy()
354 Masking
mask()

355 Image 5: Image Processing
356 Pixels
pixels[], loadPixels(), updatePixels(), createImage()
359 Pixel components
360 Convolution
364 Image as data

367 Output 1: Images
368 Saving images
save()
369 Saving sequential images
saveFrame()

371 Synthesis 3: Motion and Arrays
372 Centipede. Ariel Malka
373 Chronodraw. Andreas Gysin
374 AmoebaAbstract_03. Marius Watz
375 Mr. Roboto. Leon Hong

377 Interviews 3: Animation, Video
379 Motion Theory. *R.E.M. "Animal"*
383 Bob Sabiston. *Waking Life*
387 Jennifer Steinkamp. *Eye Catching*
391 Semiconductor. *The Mini-Epoch Series*

395 Structure 4: Objects I
395 Object-oriented programming
398 Using classes and objects
class, Object
406 Arrays of objects
409 Multiple files

413 Drawing 2: Kinetic Forms
414 Active tools
416 Active drawings

421 Output 2: File Export
421 Formatting data
nf()
422 Exporting files
saveStrings(), PrintWriter, createWriter(), PrintWriter.flush(), PrintWriter.close(), exit()

427 Input 6: File Import
428 Loading numbers
loadStrings(), split(), splitTokens()
431 Loading characters
WHITESPACE

435 Input 7: Interface
436 Rollover, Button, Dragging
442 Check boxes, Radio buttons
448 Scrollbar

453 Structure 5: Objects II
453 Multiple constructors
454 Composite objects
456 Inheritance
extends, super

461 Simulate 1: Biology
461 Cellular automata
469 Autonomous agents

477 Simulate 2: Physics
477 Motion simulation
481 Particle systems
487 Springs

495 Synthesis 4: Structure, Interface
496 WithoutTitle. Lia
497 Pond. William Ngan
498 Swingtree. ART+COM, Andreas Schlegel
499 SodaProcessing. Ed Burton

501 Interviews 4: Performance, Installation
503 SUE.C. *Mini Movies*
507 Chris Csikszentmihályi. *DJ I, Robot Sound System*
511 Golan Levin, Zachary Lieberman. *Messa di Voce*
515 Marc Hansen. *Listening Post*

519 Extension 1: Continuing…
519 Extending Processing
521 Processing and Java
522 Other programming languages

525 Extension 2: 3D. Simon Greenwold
525 A short history of 3D software
526 3D form
531 Camera
532 Material and lights
536 Tools for 3D
538 Conclusion
539 Code
545 Resources

547 Extension 3: Vision. Golan Levin
547 Computer vision in interactive art
549 Elementary computer vision techniques
552 Computer vision in the physical world
554 Tools for computer vision
555 Conclusion
556 Code
561 Resources

563 Extension 4: Network. Alexander R. Galloway
563 The Internet and the arts
565 Internet protocols and concepts
569 Network tools
571 Conclusion
572 Code
576 Resources

579 Extension 5: Sound. R. Luke DuBois
579 Music and sound programming in the arts
582 Sound and musical informatics
584 Digital representation of sound and music
588 Music as information
591 Tools for sound programming
592 Conclusion
593 Code
599 Resources

603 Extension 6: Print. Casey Reas
603 Print and computers
606 High-resolution file export
608 Production
612 Conclusion
613 Code
615 Resources

617 Extension 7: Mobile. Francis Li
617 Mobile software applications
619 The mobile platform
622 Programming for mobile phones
624 Mobile programming platforms
625 Conclusion
626 Code
631 Resources

633 Extension 8: Electronics. Hernando Barragán and Casey Reas
633 Electronics in the arts
635 Electricity
637 Components
638 Circuits
639 Microcontrollers and I/O boards
642 Sensors and communication
646 Controlling physical media
648 Conclusion
649 Code
658 Resources

661 Appendix A: Order of Operations
663 Appendix B: Reserved Words
664 Appendix C: ASCII, Unicode
669 Appendix D: Bit, Binary, Hex
673 Appendix E: Optimization
679 Appendix F: Programming Languages
686 Appendix G: Code Comparison

693 Related Media
699 Glossary
703 Code Index
705 Index

Foreword

At MIT, the full-time graduate studio that I administer attracts a uniquely gifted lot: people who have a fundamental balance issue in the way they approach the computer as an expressive medium. On the one hand, they don't want the programming code to get in the way of their designs or artistic desires; on the other hand, without hesitation they write sophisticated computer codes to discover new visual pathways. The two sides of their minds are in continual conflict. The conclusion is simple for them. Do both.

Hybrids that can fluidly cross the chasm between technology and the arts are mutations in the academic system. Traditionally, universities create technology students or art students—but never mix the two sides of the equation in the same person. During the 1990s the mutants that managed to defy this norm would either seek me out, or else I would reach out to find them myself. Bringing these unique people together was my primary passion, and that's how I came into contact with Casey Reas and Ben Fry.

It is said that the greatest compliment to a teacher is when the student surpasses the teacher. This corner was turned quickly after I began to work with them, and the finishing blow came when Ben and Casey created Processing. They prominently elevated the call for visual experimentation with their timely mastery of the Internet to engage at first tens, hundreds, and then tens of thousands of hybrids all over the world. Wherever I might travel, young technology artists are always talking about Processing and ask me to pass on their thanks to Casey and Ben.

So it is here that I express my thanks to you, Ben and Casey. On behalf of all of the people who follow where Processing might take the field of computational art and design, I wish you more sleepless nights in the relentless pursuit of perfecting the bridge that connects the art-mind with the computer-mind. All of us look to you to lead the way for when art on the computer becomes simply, art—without the icky technology connotations. We're all counting on you to take us there. Please?

John Maeda
Allen Professor of Media Arts and Sciences
MIT Media Laboratory

Preface

This book was written as an introduction to the ideas of computer programming within the context of the visual arts. It targets an audience of computer-savvy individuals who are interested in creating interactive and visual work through writing software but have little or no prior experience. We're tremendously excited about the potential of software as a medium for communication and expression, and we hope this book will open the potential to a wide audience.

The Processing book is the result of six years of software development and teaching experience. The ideas presented have been continually tested in the classrooms, computer labs, and basements of universities, art and design schools, and arts institutions. The authors have taught related courses at the University of California–Los Angeles, the Interaction Design Institute Ivrea, Harvard University, and Carnegie Mellon University and have given numerous workshops and lectures on this topic at conferences and institutions around the globe. The contents of this book have been continually improved through the generous feedback of students and fellow educators. The refined curriculum is presented here in book form with the intention of distributing the results of this endeavor to a larger and more diverse community.

Contents

Four types of content are featured in these pages. The majority of the book is divided into tutorial units discussing specific elements of software and how they relate to the arts. These units introduce the syntax and concepts of software such as variables, functions, and object-oriented programming. They cover topics such as photography and drawing in relation to software. These units feature many short, prototypical example programs with related images and explanation. More advanced professional projects from diverse domains including animation, performance, and typography are discussed in interviews with their creators (pp. 155, 261, 377, 501). The extension sections (p. 519) present concise introductions to further domains of exploration including computer vision, sound, and electronics. The appendixes (p. 661) provide reference tables and more involved explanations of technical topics. The related media section (p. 693) is a list of references to additional material on related topics. The technical terms used in the book are defined in the glossary (p. 699).

This book is full of example programs written using the Processing programming language developed by the authors. Processing is a free, open source programming language and environment used by students, artists, designers, architects, researchers, and hobbyists for learning, prototyping, and production. Processing is developed by artists and designers as an alternative to proprietary software tools in the same domain. The project integrates a programming language, development environment,

and teaching methodology into a unified structure for learning and exploration. The software allows people to make a smooth transition from beginner to advanced programmer, and the Processing language is a good basis for future learning. The technical aspects of the language and the deeper programming concepts introduced in this text translate well to other programming languages, particularly those used frequently within the arts.

Most of the examples presented in this book have a minimal visual style. This represents not a limitation of the Processing software, but rather a conscious decision by the authors to make the code for each example as brief and clear as possible. We hope the stark quality of the examples gives additional incentive to the reader to extend the programs to her or his own visual language.

How to read this book

This book can be read front to back or in a self-directed order. There are two tables of contents (p. vii, ix) that order the book in different ways. In addition to reading the book from front to back, following each category (e.g., Input, Shape, Structure) from beginning to end is a logical way to move through the text. Previous knowledge and areas of interest can define the order of exploration. For example, it's possible to read all of the units about images and skip those about math, or vice versa. You will find that many later units require knowledge of concepts discussed in earlier units. If you find unfamiliar code and ideas, it may be necessary to read earlier units before proceeding.

Understanding this book requires more than reading the words. It is also essential to run, modify, and interact with the programs found within. Just as it's not possible to learn to cook without cooking, it's not possible to learn how to program without programming. Many of the examples can be fully understood only in motion or in response to the mouse and keyboard. The Processing software and all of the code presented in this book can be downloaded and run for future exploration. Processing can be downloaded from *www.processing.org/download* and the examples from *www.processing.org/learning*.

The code, diagrams, and images convey essential content to augment the text. Because this book was written for visually oriented people, it's assumed that diagrams and images will be read with as much care as the text. Typographic and visual conventions are used to assist reading. Code elements within the text are presented in a `monospaced font` for differentiation. Each code example is numbered sequentially to make it easy to reference. The numbers appear in the right margin at the first line of each example. The number "15-02" refers to the 2nd example in the 15th unit (p. 128). Unfortunately, sometimes a code example wraps to the next page. When the abbreviation "cont." appears as a part of the code number, this signifies the code is *continued* from the previous page. Many of the code examples run differently when the variable values are changed. When numbers appear to the left of an image (p. 200), these numbers were used to produce that image. In examples where the mouse position is important, thin lines are used to imply the mouse position at the time the image was

made (code 23-02, p. 206). In some examples, only the horizontal position of the mouse is important and a single vertical line is used to show the position (code 23-03, p. 206).

Casey's introduction

I started playing with computers as a child. I played games and wrote simple programs in BASIC and Logo on my family's Apple IIe machine. I spent years exploring and testing it, but I preferred drawing, and my interest in computers dissipated. As a design student at the University of Cincinnati in the early 1990s, I started to use Adobe's Photoshop and Illustrator programs during my first year, but I wasn't permitted to use them in my design studio classes until the third year. I spent the first two years of my education training my eyes and hands to construct composition and meaning through visual form. I focused my energy on drawing icons and letters with pencils and painting them with Plaka, a matte black paint. This was intensely physical work. I often produced hundreds of paper sketches while working toward a single refined image. I later focused my efforts on the design of printed materials including books, magazines, and information graphics. In this work I used software as a tool during an intermediate stage between concept and the final result on paper.

Over time, I shifted from producing printed media to software. When the multimedia CD-ROM industry emerged, I worked in that area to integrate my interests in sound, video, image, and information design. With the rise of the Internet in the mid-1990s, I focused on building large, database-integrated websites. As I shifted my work from paper to screen, static grids and information hierarchies evolved into kinetic, modular systems with variable resolutions and compositions. The time and energy once spent devoted to details of materials and static composition shifted to details of motion and response. I focused on building real-time processes to generate form, define behavior, and mediate interactions. To pursue these interests at a more advanced level, I realized I would need to learn to program computers. After a childhood of playing with computers and years of working with them professionally, I started down a new path.

In 1997 I met John Maeda and was introduced to the experimental software work of his students in the Aesthetics and Computation Group at MIT. They created a new type of work by fusing traditional arts knowledge with ideas from computer science. My new direction emerged as I experienced this work, and in 1998 I started learning to program computers in earnest. I began graduate studies at MIT the following year. My time there was personally transforming as I shifted from a consumer of software to a producer. I expanded my views of technology in relation to culture and the history of art.

While a graduate student at the MIT Media Lab, I was introduced to a culture of individuals who combined skills from more than one field of study. The knowledge in common was computing technology, and people had backgrounds in other disciplines including architecture, art, mathematics, design, and music. At that time, few software environments afforded both a sophisticated programming language *and* the ability to created refined graphics, so my predecessors and colleagues at MIT built their own software to meet their unique needs. The building of these software tools and their use

to develop projects led to the emergence of a unique culture that synthesized knowledge from visual culture with knowledge from computer science. The desire to make this information accessible to people outside of technical fields and institutions has been my motivation for dedicating the last six years to developing Processing. I hope this book will act as a catalyst to increase software literacy within the arts.

Ben's introduction

Like lots of people who wind up in similar careers, I've always been interested in taking things apart to understand how they work. This began with disassembling and comparing the contents of household electronics, looking for similar components. Once I ran out of telephones and radios, I moved to software. The computer provided an endless range of questions, like having an infinite number of telephones. With a burnt yellow binder that described "IBM BASIC by Microsoft," my father introduced me to the "for" loop and I gradually taught myself programming—mostly by staring at others' code, sometimes modifying it to make it do something else. Over time it became easier to write code from scratch.

I had a separate interest in graphic design, and I was curious about typography and layout and composition. A family friend ran a design firm, and I thought that seemed like the most interesting job on earth. I later applied to design school, thinking of studying user interface design or creating "interactive multimedia CD-ROMs," the only possibilities I could see for intersecting my two interests. Attending design school was significant for me, because it provided thinking and creative skills that could be applied to other areas, including my interest in software.

In 1997, during my final year of undergraduate school, John Maeda gave a lecture at our program. It was overwhelming for several of us, including one friend who sat mumbling "Whoa, slow down..." as we watched from the back of the room. In the presentation I finally saw the intersection between design and computation that I couldn't figure out before. It was a different perspective than building tools, which sounded mundane, or building interfaces, which also left something to be desired. A year later I was lucky to have the opportunity to join Professor Maeda at MIT.

Pedagogy was a persistent theme during my six years working with John at the Media Laboratory. Casey, other students, and I contributed to the Design By Numbers project, which taught us a great deal about teaching computation to designers and gave us a lot of feedback on what people wanted. Casey and I began to see a similarity between this feature set and what we did in our own work at the "sketching" stage, and we started to discuss how we might connect the two in what would later be called Processing.

We wanted Processing to include lots of code that could be viewed, modified, and tested—reflecting the way in which I learned programming. But more important has been the community that has formed around the project, who are eager to share code with one another and help answer each other's questions. In a similar manner, the code for Processing itself is available, which for me has a lot to do with repaying the favor of

a previous generation of developers who shared their code and answered my questions.

One of my personal goals for this project is to facilitate designers' taking control of their own tools. It's been more than twenty years since desktop publishing helped reinvent design in the mid-1980s, and we're overdue for more innovations. As designers have become fed up with available tools, coding and scripting have begun to fill the widening gap between what's in the designer's mind and the capability of the software they've purchased. While most users of Processing will apply it to their own work, I hope that it will also enable others to create new design tools that come not from corporations or computer scientists, but from designers themselves.

Acknowledgments

This book is the synthesis of more than fifteen years of studying visual design and software. John Maeda is the person most responsible for the genesis of Processing and this book. His guidance as our adviser in the Aesthetics and Computation Group (ACG) at the MIT Media Lab and the innovations of the Design By Numbers project are the foundation for the ideas presented here. Processing has also been strongly informed by the research and collaboration with our fellow graduate students at the ACG from 1999 through 2004. We are grateful for our collaboration with Peter Cho, Elise Co, Megan Galbraith, Simon Greenwold, Omar Khan, Axel Kilian, Reed Kram, Golan Levin, Justin Manor, Nikita Pashenkov, Jared Schiffman, David Small, and Tom White. We also acknowledge the foundation built by our predecessors in the ACG and the Visual Language Workshop.

While Processing's origin is at MIT, it has grown up within a set of other institutions including UCLA, the Interaction Design Institute Ivrea, the Broad Institute, and Carnegie Mellon. Casey's colleagues in Los Angeles and Ivrea have provided the environment for many of the ideas in this text to evolve. We thank UCLA faculty members Rebecca Allen, Mark Hansen, Erkki Huhtamo, Robert Israel, Willem Henri Lucas, Rebeca Mendez, Vasa Mihich, Christian Moeller, Jennifer Steinkamp, and Victoria Vesna. We thank Ivrea faculty members and founders Gillian Crampton-Smith, Andrew Davidson, Dag Svanaes,Walter Aprile, Michael Kieslinger, Stefano Mirti, Jan-Christoph Zoels, Massimo Banzi, Nathan Shedroff, Bill Moggridge, John Thackara, and Bill Verplank. From the Broad Institute we thank Eric Lander for funding Ben Fry's visualization research, most of which was built with Processing.

The ideas and structure of this book have been refined over the last six years of teaching at UCLA, Carnegie Mellon, Interaction Design Institute Ivrea, MIT, and Harvard. We're particularly grateful to the students in Casey's DESMA 28, 152A, and 152B classes for their ideas, effort, and energy. Casey's graduate students at UCLA have provided invaluable feedback: Tatsuya Saito, Krister Olsson, Aaron Koblin, John Houck, Zai Chang, and Andrew Hieronomi.

Processing was first introduced to students through workshops. We're eternally grateful to the first institutions that took a chance on our new software in 2001 and 2002: Musashino Art University (Tokyo), ENSCI – Les Ateliers (Paris), HyperWerk (Basel),

and the Royal Conservatory (Hague). Many universities have integrated Processing into their curriculum, and we're grateful to the pioneer faculty members and students at these institutions. They are too numerous to mention here. The students and faculty in New York University's Interactive Telecommunication Program (ITP) deserve a special thank you for their early adoption and promotion, particularly Dan O'Sullivan, Josh Nimoy, Amit Pitaru, and Dan Shiffman.

The Processing software is a community effort. Over the last five years, the software has evolved through a continuous conversation. The goals of this book and of the Processing software have expanded and contracted as a result of invaluable suggestions and lively debates. It's impossible to make a list of everyone who has collaborated and contributed. The people who have formally contributed to the software include Karsten Schmidt, Ariel Malka, Martin Gomez, Mikkel Crone Koser, Koen Mostert, Timothy Mohn, Dan Mosedale, Jacob Schwartz, Sami Arola, Dan Haskovec, and Jonathan Feinberg. The website *www.processing.org* has been augmented by Lenny Burdette, Florian Jenett, Cem Uzunoglu, Dara Kilicoglu, and Kevin Cannon. The Processing reference (found on the website) has been translated into other languages by Willian Ngan, Tori Tan, Mei Yu, Widianto Nugroho, Tetsu Kondo, Tai-Kyung Kim, Julien Gachadoat, Pedro Alpera, Alessandro Capozzo, and Burak Arikan. As of 30 June 2006, code libraries have been generously contributed by Brendan Berg, Jeffrey Traer Bernstein, Michael Chang, Stephane Cousot, Jeff Crouse, Kristian Linn Damkjer, Daniel Dihardja, Julien Gachadoat, Simon Greenwold, Mark Hill, Florian Jenett, JohnG, Jesse Kriss, Ariel Malka, Markavian, Allan William Martin, Josh Nimoy, Krister Olsson, Amit Pitaru, Christian Riekoff, RSG, Carl-Johan Rosén, Tatsuya Saito, Andreas Schlegel, Karsten Schmidt, Daniel Shiffman, Taka, and Marius Watz. Tom Carden created *www.processingblogs.org* and together with Karsten Schmidt created *www.processinghacks.com*.

We also thank the open source software developers of Jikes, JEdit, ORO Matcher, and ANTLR. The Processing software is built upon their work.

This text has been rewritten and redesigned countless times over the last two years. We're indebted to Shannon Hunt, who read and edited the first draft and also proofread the final manuscript. Karsten Schmidt and Larry Cuba read early chapters and provided feedback. Tom Igoe and David Cuartielles provided essential feedback for the Electronics extension and Drew Trujillo helped immensely with Appendix G. Rajorshi Ghosh and Mary Huang provided invaluable production assistance. We're very grateful to Chandler McWilliams for executing a thorough technical review of the final manuscript.

We've enjoyed working with the folks at MIT Press and we thank them for their dedication to this project. Doug Sery has guided us through every step of the publication process and that has made this book possible. We thank Katherine Almeida and the editorial staff for minding our Ps and Qs and we are grateful for the production wisdom of Terry Lamoureux and Jennifer Flint. The anonymous reviewers of the proposal and draft provided extremely valuable feedback that helped to refine the book's structure.

We thank the many contributing artists and authors. They were generous with their time, and this book is greatly enhanced through their efforts.

Most importantly, Casey thanks Cait, Molly, Bob, and Deanna. Ben thanks Shannon, Chief, Rose, Mimi, Jamie, Leif, Erika, and Josh.

Processing...

Processing relates software concepts to principles of visual form, motion, and interaction. It integrates a programming language, development environment, and teaching methodology into a unified system. Processing was created to teach fundamentals of computer programming within a visual context, to serve as a software sketchbook, and to be used as a production tool. Students, artists, design professionals, and researchers use it for learning, prototyping, and production.

The Processing language is a text programming language specifically designed to generate and modify images. Processing strives to achieve a balance between clarity and advanced features. Beginners can write their own programs after only a few minutes of instruction, but more advanced users can employ and write libraries with additional functions. The system facilitates teaching many computer graphics and interaction techniques including vector/raster drawing, image processing, color models, mouse and keyboard events, network communication, and object-oriented programming. Libraries easily extend Processing's ability to generate sound, send/receive data in diverse formats, and to import/export 2D and 3D file formats.

Software

A group of beliefs about the software medium set the conceptual foundation for Processing and inform decisions related to designing the software and environment.

Software is a unique medium with unique qualities
Concepts and emotions that are not possible to express in other media may be expressed in this medium. Software requires its own terminology and discourse and should not be evaluated in relation to prior media such as film, photography, and painting. History shows that technologies such as oil paint, cameras, and film have changed artistic practice and discourse, and while we do not claim that new technologies improve art, we do feel they enable different forms of communication and expression. Software holds a unique position among artistic media because of its ability to produce dynamic forms, process gestures, define behavior, simulate natural systems, and integrate other media including sound, image, and text.

Every programming language is a distinct material
As with any medium, different materials are appropriate for different tasks. When designing a chair, a designer decides to use steel, wood or other materials based on the intended use and on personal ideas and tastes. This scenario transfers to writing software. The abstract animator and programmer Larry Cuba describes his experience this way: "Each of my films has been made on a different system using a different

programming language. A programming language gives you the power to express some ideas, while limiting your abilities to express others."[1] There are many programming languages available from which to choose, and some are more appropriate than others depending on the project goals. The Processing language utilizes a common computer programming syntax that makes it easy for people to extend the knowledge gained through its use to many diverse programming languages.

Sketching is necessary for the development of ideas
It is necessary to sketch in a medium related to the final medium so the sketch can approximate the finished product. Painters may construct elaborate drawings and sketches before executing the final work. Architects traditionally work first in cardboard and wood to better understand their forms in space. Musicians often work with a piano before scoring a more complex composition. To sketch electronic media, it's important to work with electronic materials. Just as each programming language is a distinct material, some are better for sketching than others, and artists working in software need environments for working through their ideas before writing final code. Processing is built to act as a software sketchbook, making it easy to explore and refine many different ideas within a short period of time.

Programming is not just for engineers
Many people think programming is only for people who are good at math and other technical disciplines. One reason programming remains within the domain of this type of personality is that the technically minded people usually create programming languages. It is possible to create different kinds of programming languages and environments that engage people with visual and spatial minds. Alternative languages such as Processing extend the programming space to people who think differently. An early alternative language was Logo, designed in the late 1960s by Seymour Papert as a language concept for children. Logo made it possible for children to program many different media, including a robotic turtle and graphic images on screen. A more contemporary example is the Max programming environment developed by Miller Puckette in the 1980s. Max is different from typical languages; its programs are created by connecting boxes that represent the program code, rather than lines of text. It has generated enthusiasm from thousands of musicians and visual artists who use it as a base for creating audio and visual software. The same way graphical user interfaces opened up computing for millions of people, alternative programming environments will continue to enable new generations of artists and designers to work directly with software. We hope Processing will encourage many artists and designers to tackle software and that it will stimulate interest in other programming environments built for the arts.

Literacy

Processing does not present a radical departure from the current culture of programming. It repositions programming in a way that is accessible to people who are interested in programming but who may be intimidated by or uninterested in the type taught in computer science departments. The computer originated as a tool for fast calculations and has evolved into a medium for expression.

The idea of general software literacy has been discussed since the early 1970s. In 1974, Ted Nelson wrote about the minicomputers of the time in *Computer Lib / Dream Machines*. He explained "the more you know about computers . . . the better your imagination can flow between the technicalities, can slide the parts together, can discern the shapes of what you would have these things do."[2] In his book, Nelson discusses potential futures for the computer as a media tool and clearly outlines ideas for hypertexts (linked text, which set the foundation for the Web) and hypergrams (interactive drawings). Developments at Xerox PARC led to the Dynabook, a prototype for today's personal computers. The Dynabook vision included more than hardware. A programming language was written to enable, for example, children to write storytelling and drawing programs and musicians to write composition programs. In this vision there was no distinction between a computer user and a programmer.

Thirty years after these optimistic ideas, we find ourselves in a different place. A technical and cultural revolution did occur through the introduction of the personal computer and the Internet to a wider audience, but people are overwhelmingly using the software tools created by professional programmers rather than making their own. This situation is described clearly by John Maeda in his book *Creative Code*: "To use a tool on a computer, you need do little more than point and click; to create a tool, you must understand the arcane art of computer programming."[3] The negative aspects of this situation are the constraints imposed by software tools. As a result of being easy to use, these tools obscure some of the computer's potential. To fully explore the computer as an artistic material, it's important to understand this "arcane art of computer programming."

Processing strives to make it possible and advantageous for people within the visual arts to learn how to build their own tools—to become software literate. Alan Kay, a pioneer at Xerox PARC and Apple, explains what literacy means in relation to software:

> The ability to "read" a medium means you can access materials and tools created by others. The ability to "write" in a medium means you can generate materials and tools for others. You must have both to be literate. In print writing, the tools you generate are rhetorical; they demonstrate and convince. In computer writing, the tools you generate are processes; they simulate and decide.[4]

Making processes that simulate and decide requires programming.

Open

The open source software movement is having a major impact on our culture and economy through initiatives such as Linux, but it is having a smaller influence on the culture surrounding software for the arts. There are scattered small projects, but companies such as Adobe and Microsoft dominate software production and therefore control the future of software tools used within the arts. As a group, artists and designers traditionally lack the technical skills to support independent software initiatives. Processing strives to apply the spirit of open source software innovation to the domain of the arts. We want to provide an alternative to available proprietary software and to improve the skills of the arts community, thereby stimulating interest in related initiatives. We want to make Processing easy to extend and adapt and to make it available to as many people as possible.

Processing probably would not exist without its ties to open source software. Using existing open source projects as guidance, and for important software components, has allowed the project to develop in a smaller amount of time and without a large team of programmers. Individuals are more likely to donate their time to an open source project, and therefore the software evolves without a budget. These factors allow the software to be distributed without cost, which enables access to people who cannot afford the high prices of commercial software. The Processing source code allows people to learn from its construction and by extending it with their own code.

People are encouraged to publish the code for programs they've written in Processing. The same way the "view source" function in Web browsers encouraged the rapid proliferation of website-creation skills, access to others' Processing code enables members of the community to learn from each other so that the skills of the community increase as a whole. A good example involves writing software for tracking objects in a video image, thus allowing people to interact directly with the software through their bodies, rather than through a mouse or keyboard. The original submitted code worked well but was limited to tracking only the brightest object in the frame. Karsten Schmidt (a k a toxi), a more experienced programmer, used this code as a foundation for writing more general code that could track multiple colored objects at the same time. Using this improved tracking code as infrastructure enabled Laura Hernandez Andrade, a graduate student at UCLA, to build *Talking Colors*, an interactive installation that superimposes emotive text about the colors people are wearing on top of their projected image. Sharing and improving code allows people to learn from one another and to build projects that would be too complex to accomplish without assistance.

Education

Processing makes it possible to introduce software concepts in the context of the arts and also to open arts concepts to a more technical audience. Because the Processing syntax is derived from widely used programming languages, it's a good base for future learning. Skills learned with Processing enable people to learn other programming

languages suitable for different contexts including Web authoring, networking, electronics, and computer graphics.

There are many established curricula for computer science, but by comparison there have been very few classes that strive to integrate media arts knowledge with core concepts of computation. Using classes initiated by John Maeda as a model, hybrid courses based on Processing are being created. Processing has proved useful for short workshops ranging from one day to a few weeks. Because the environment is so minimal, students are able to begin programming after only a few minutes of instruction. The Processing syntax, similar to other common languages, is already familiar to many people, and so students with more experience can begin writing advanced syntax almost immediately.

In a one-week workshop at Hongik University in Seoul during the summer of 2003, the students were a mix of design and computer science majors, and both groups worked toward synthesis. Some of the work produced was more visually sophisticated and some more technically advanced, but it was all evaluated with the same criteria. Students like Soo-jeong Lee entered the workshop without any previous programming experience; while she found the material challenging, she was able to learn the basic principles and apply them to her vision. During critiques, her strong visual skills set an example for the students from more technical backgrounds. Students such as Tai-kyung Kim from the computer science department quickly understood how to use the Processing software, but he was encouraged by the visuals in other students' work to increase his aesthetic sensibility. His work with kinetic typography is a good example of a synthesis between his technical skills and emerging design sensitivity.

Processing is also used to teach longer introductory classes for undergraduates and for topical graduate-level classes. It has been used at small art schools, private colleges, and public universities. At UCLA, for example, it is used to teach a foundation class in digital media to second-year undergraduates and has been introduced to the graduate students as a platform for explorations into more advanced domains. In the undergraduate Introduction to Interactivity class, students read and discuss the topic of interaction and make many examples of interactive systems using the Processing language. Each week new topics such as kinetic art and the role of fantasy in video games are introduced. The students learn new programming skills, and they produce an example of work addressing a topic. For one of their projects, the students read Sherry Turkle's "Video Games and Computer Holding Power"[5] and were given the assignment to write a short game or event exploring their personal desire for escape or transformation. Leon Hong created an elegant flying simulation in which the player floats above a body of water and moves toward a distant island. Muskan Srivastava wrote a game in which the objective was to consume an entire table of desserts within ten seconds.

Teaching basic programming techniques while simultaneously introducing basic theory allows the students to explore their ideas directly and to develop a deep understanding and intuition about interactivity and digital media. In the graduate-level Interactive Environments course at UCLA, Processing is used as a platform for experimentation with computer vision. Using sample code, each student has one week to develop software that uses the body as an input via images from a video camera.

Zai Chang developed a provocative installation called White Noise where participants' bodies are projected as a dense series of colored particles. The shadow of each person is displayed with a different color, and when they overlap, the particles exchange, thus appearing to transfer matter and infect each other with their unique essence. Reading information from a camera is an extremely simple action within the Processing environment, and this facility fosters quick and direct exploration within courses that might otherwise require weeks of programming tutorials to lead up to a similar project.

Network

Processing takes advantage of the strengths of Web-based communities, and this has allowed the project to grow in unexpected ways. Thousands of students, educators, and practitioners across five continents are involved in using the software. The project website serves as the communication hub, but contributors are found remotely in cities around the world. Typical Web applications such as bulletin boards host discussions between people in remote locations about features, bugs, and related events.

Processing programs are easily exported to the Web, which supports networked collaboration and individuals sharing their work. Many talented people have been learning rapidly and publishing their work, thus inspiring others. Websites such as Jared Tarbell's *Complexification.net* and Robert Hodgin's *Flight404.com* present explorations into form, motion, and interaction created in Processing. Tarbell creates images from known algorithms such as Henon Phase diagrams and invents his own algorithms for image creation, such as those from *Substrate*, which are reminiscent of urban patterns (p. 157). On sharing his code from his website, Tarbell writes, "Opening one's code is a beneficial practice for both the programmer and the community. I appreciate modifications and extensions of these algorithms."[6] Hodgin is a self-trained programmer who uses Processing to explore the software medium. It has allowed him to move deeper into the topic of simulating natural forms and motion than he could in other programming environments, while still providing the ability to upload his software to the Internet. His highly trafficked website documents these explorations by displaying the running software as well as providing supplemental text, images, and movies. Websites such as those developed by Jared and Robert are popular destinations for younger artists and designers and other interested individuals. By publishing their work on the Web in this manner they gain recognition within the community.

Many classes taught using Processing publish the complete curriculum on the Web, and students publish their software assignments and source code from which others can learn. The websites for Daniel Shiffman's classes at New York University, for example, include online tutorials and links to the students' work. The tutorials for his Procedural Painting course cover topics including modular programming, image processing, and 3D graphics by combining text with running software examples. Each student maintains a web page containing all of their software and source code created for the class. These pages provide a straightforward way to review performance and make it easy for members of the class to access each others's work.

The Processing website, *www.processing.org*, is a place for people to discuss their projects and share advice. The Processing Discourse section of the website, an online bulletin board, has thousands of members, with a subset actively commenting on each others' work and helping with technical questions. For example, a recent post focused on a problem with code to simulate springs. Over the course of a few days, messages were posted discussing the details of Euler integration in comparison to the Runge-Kutta method. While this may sound like an arcane discussion, the differences between the two methods can be the reason a project works well or fails. This thread and many others like it are becoming concise Internet resources for students interested in detailed topics.

Context

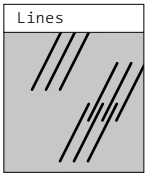
The Processing approach to programming blends with established methods. The core language and additional libraries make use of Java, which also has elements identical to the C programming language. This heritage allows Processing to make use of decades of programming language refinements and makes it understandable to many people who are already familiar with writing software.

Processing is unique in its emphasis and in the tactical decisions it embodies with respect to its context within design and the arts. Processing makes it easy to write software for drawing, animation, and reacting to the environment, and programs are easily extended to integrate with additional media types including audio, video, and electronics. Modified versions of the Processing environment have been built by community members to enable programs to run on mobile phones (p. 617) and to program microcontrollers (p. 633).

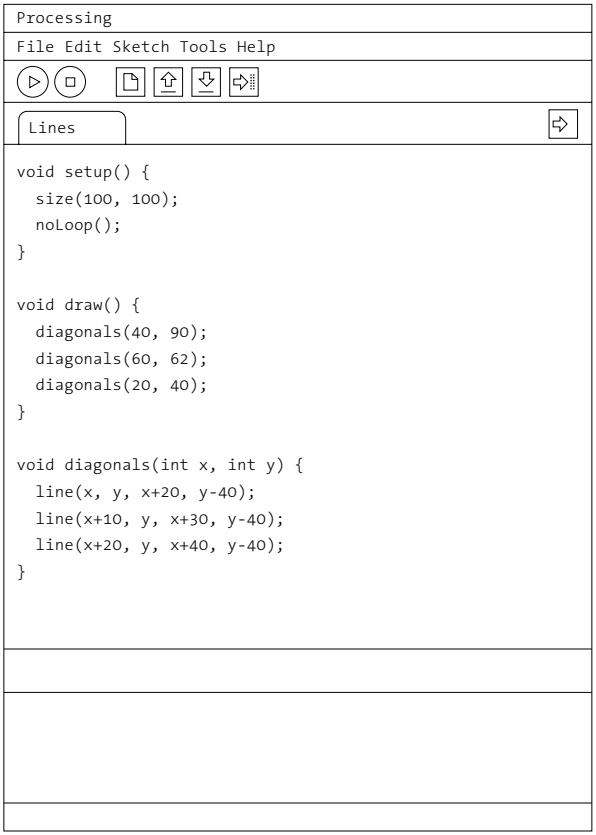
The network of people and schools using the software continues to grow. In the five years since the origin on the idea for the software, it has evolved organically through presentations, workshops, classes, and discussions around the globe. We plan to continually improve the software and foster its growth, with the hope that the practice of programming will reveal its potential as the foundation for a more dynamic media.

Notes

1. Larry Cuba, "Calculated Movements," in *Prix Ars Electronica Edition '87: Meisterwerke der Computerkunst* (H. S. Sauer, 1987), p. 111.
2. Theodore Nelson, "Computer Lib / Dream Machines," in *The New Media Reader*, edited by Noah Wardrip-Fruin and Nick Montfort (MIT Press, 2003), p. 306.
3. John Maeda, *Creative Code* (Thames & Hudson, 2004), p. 113.
4. Alan Kay, "User Interface: A Personal View," in *The Art of Human-Computer Interface Design*, edited by Brenda Laurel (Addison-Wesley, 1989), p. 193.
5. Chapter 2 in Sherry Turkle, *The Second Self: Computers and the Human Spirit* (Simon & Schuster, 1984), pp. 64–92.
6. Jared Tarbell, Complexification.net (2004), http://www.complexification.net/medium.html.



Display window



Menu

Toolbar

Tabs

Text editor

Message area

Console

Processing Development Environment (PDE)

Use the PDE to create programs. Write the code in the text editor and use the buttons in the toolbar to run, save, and export the code.

Using Processing

Download, Install

The Processing software can be downloaded from the Processing website. Using a Web browser, navigate to *www.processing.org/download* and click on the link for your computer's operating system. The Processing software is available for Linux, Macintosh, and Windows. The most up-to-date installation instructions for your operating system are linked from this page.

Environment

The Processing Development Environment (PDE) consists of a simple text editor for writing code, a message area, a text console, tabs for managing files, a toolbar with buttons for common actions, and a series of menus. When programs are run, they open in a new window called the display window.

Pieces of software written using Processing are called sketches. These sketches are written in the text editor. It has features for cutting/pasting and for searching/replacing text. The message area gives feedback while saving and exporting and also displays errors. The console displays text output by Processing programs including complete error messages and text output from programs with the `print()` and `println()` functions. The toolbar buttons allow you to run and stop programs, create a new sketch, open, save, and export.

Run	Compiles the code, opens a display window, and runs the program inside.
Stop	Terminates a running program, but does not close the display window.
New	Creates a new sketch.
Open	Provides a menu with options to open files from the sketchbook, open an example, or open a sketch from anywhere on your computer or network.
Save	Saves the current sketch to its current location. If you want to give the sketch a different name, select "Save As" from the File menu.
Export	Exports the current sketch as a Java applet embedded in an HTML file. The folder containing the files is opened. Click on the *index.html* file to load the software in the computer's default Web browser.

The menus provide the same functionality as the toolbar in addition to actions for file management and opening reference materials.

File	Commands to manage and export files
Edit	Controls for the text editor (Undo, Redo, Cut, Copy, Paste, Find, Replace, etc.)

Sketch	Commands to run and stop programs and to add media files and code libraries.
Tools	Tools to assist in using Processing (automated code formatting, creating fonts, etc.)
Help	Reference files for the environment and language

All Processing projects are called sketches. Each sketch has its own folder. The main program file for each sketch has the same name as the folder and is found inside. For example, if the sketch is named *Sketch_123*, the folder for the sketch will be called *Sketch_123* and the main file will be called *Sketch_123.pde*. The PDE file extension stands for the Processing Development Environment.

A sketch folder sometimes contains other folders for media files and code libraries. When a font or image is added to a sketch by selecting "Add File" from the Sketch menu, a *data* folder is created. You can also add files to your Processing sketch by dragging them into the text editor. Image and sound files dragged into the application window will automatically be added to the current sketch's *data* folder. All images, fonts, sounds, and other data files loaded in the sketch must be in this folder. Sketches are stored in the Processing folder, which will be in different places on your computer or network depending on whether you use PC, Mac, or Linux and on how the preferences are set. To locate this folder, select the "Preferences" option from the File menu (or from the Processing menu on the Mac) and look for the "Sketchbook location."

It is possible to have multiple files in a single sketch. These can be Processing text files (with the extension *.pde*) or Java files (with the extension *.java*). To create a new file, click on the arrow button to the right of the file tabs. This button enables you to create, delete, and rename the files that comprise the current sketch. You can write functions and classes in new PDE files and you can write any Java code in files with the JAVA extension. Working with multiple files makes it easier to reuse code and to separate programs into small subprograms. This is discussed in more detail in Structure 4 (p. 395).

Export

The export feature packages a sketch to run within a Web browser. When code is exported from Processing it is converted into Java code and then compiled as a Java applet. When a project is exported, a series of files are written to a folder named *applet* that is created within the sketch folder. All files from the sketch folder are exported into a single Java Archive (JAR) file with the same name as the sketch. For example, if the sketch is named *Sketch_123*, the exported file will be called *Sketch_123.jar*. The *applet* folder contains the following:

index.html	HTML file with the applet embedded and a link to the source code and the Processing homepage. Double-click this file to open it in the default Web browser.
Sketch_123.jar	Java Archive containing all necessary files for the sketch to run. Includes the Processing core classes, those written for the sketch, and all included media files from the data folder such as images, fonts, and sounds.

Sketch_123.java	The JAVA file generated by the preprocessor from the PDE file. This is the actual file that is compiled into the applet by the Java compiler used in Processing.
Sketch_123.pde	The original program file. It is linked from the index.html file.
loading.gif	An image file displayed while the program is loading in a Web browser.

Every time a sketch is exported, the contents of the *applet* folder are deleted and the files are written from scratch. Any changes previously made to the *index.html* file are lost. Media files not needed for the applet should be deleted from the *data* folder before it is exported to keep the file size small. For example, if there are unused images in the *data* folder, they will be added to the JAR file, thus needlessly increasing its size.

In addition to exporting Java applets for the Web, Processing can also export Java applications for the Linux, Macintosh, and Windows platforms. When "Export Application" is selected from the File menu, folders will be created for each of the operating systems specified in the Preferences. Each folder contains the application, the source code for the sketch, and all required libraries for a specific platform.

Additional and updated information about the Processing environment is available at *www.processing.org/reference/environment* or by selecting the "Environment" item from the Help menu of the Processing application.

Example walk-through

A Processing program can be be as short as one line of code and as long as thousands of lines. This scalability is one of the most important aspects of the language. The following example walk-through presents the modest goal of animating a sequence of diagonal lines as a means to explore some of the basic components of the Processing language. If you are new to programming, some of the terminology and symbols in this section will be unfamiliar. This walk-through is a condensed overview of the entire book, utilizing ideas and techniques that are covered in detail later. Try running these programs inside the Processing application to better understand what the code is doing.

Processing was designed to make it easy to draw graphic elements such as lines, ellipses, and curves in the display window. These shapes are positioned with numbers that define their coordinates. The position of a line is defined by four numbers, two for each endpoint. The parameters used inside the `line()` function determine the position where the line appears. The origin of the coordinate system is in the upper-left corner, and numbers increase right and down. Coordinates and drawing different shapes are discussed on pages 23–30.

```
line(10, 80, 30, 40);  // Left line
line(20, 80, 40, 40);
line(30, 80, 50, 40);  // Middle line
line(40, 80, 60, 40);
line(50, 80, 70, 40);  // Right line
```

0-01

The visual attributes of shapes are controlled with other code elements that set color and gray values, the width of lines, and the quality of the rendering. Drawing attributes are discussed on pages 31–35.

0-02

```
background(0);          // Set the black background
stroke(255);            // Set line value to white
strokeWeight(5);        // Set line width to 5 pixels
smooth();               // Smooth line edges
line(10, 80, 30, 40);   // Left line
line(20, 80, 40, 40);
line(30, 80, 50, 40);   // Middle line
line(40, 80, 60, 40);
line(50, 80, 70, 40);   // Right line
```

A variable, such as `x`, represents a value; this value replaces the symbol `x` when the code is run. One variable can then control many features of the program. Variables are introduced on page 37-41.

0-03

```
int x = 5;   // Set the horizontal position
int y = 60;  // Set the vertical position
line(x, y, x+20, y-40);    // Line from [5,60] to [25,20]
line(x+10, y, x+30, y-40); // Line from [15,60] to [35,20]
line(x+20, y, x+40, y-40); // Line from [25,60] to [45,20]
line(x+30, y, x+50, y-40); // Line from [35,60] to [55,20]
line(x+40, y, x+60, y-40); // Line from [45,60] to [65,20]
```

Adding more structure to a program opens further possibilities. The `setup()` and `draw()` functions make it possible for the program to run continuously—this is required to create animation and interactive programs. The code inside `setup()` runs once when the program first starts, and the code inside `draw()` runs continuously. One image frame is drawn to the display window at the end of each loop through `draw()`.

In the following example, the variable `x` is declared as a global variable, meaning it can be assigned and accessed anywhere in the program. The value of `x` increases by 1 each frame, and because the position of the lines is controlled by `x`, they are drawn to a different location each time the value changes. This moves the lines to the right.

Line 14 in the code is an `if` structure. It contains a relational expression comparing the variable `x` to the value 100. When the expression is `true`, the code inside the block (the code between the { and } associated with the `if` structure) runs. When the relational expression is `false`, the code inside the block does not run. When the value of `x` becomes greater than 100, the line of code inside the block sets the variable `x` to -40, causing the lines to jump to the left edge of the window. The details of `draw()` are discussed on pages 173–175, programming animation is discussed on pages 315–320, and the `if` structure is discussed on pages 53–56.

```
int x = 0;   // Set the horizontal position
int y = 55;  // Set the vertical position

void setup() {
  size(100, 100);  // Set the window to 100 x 100 pixels
}

void draw() {
  background(204);
  line(x, y, x+20, y-40);     // Left line
  line(x+10, y, x+30, y-40);  // Middle line
  line(x+20, y, x+40, y-40);  // Right line
  x = x + 1;      // Add 1 to x
  if (x > 100) {  // If x is greater than 100,
    x = -40;      // assign -40 to x
  }
}
```

0-04

When a program is running continuously, Processing stores data from input devices such as the mouse and keyboard. This data can be used to affect what is happening in the display window. Programs that respond to the mouse are discussed on pages 205–244.

```
void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  // Assign the horizontal value of the cursor to x
  float x = mouseX;
  // Assign the vertical value of the cursor to y
  float y = mouseY;
  line(x, y, x+20, y-40);
  line(x+10, y, x+30, y-40);
  line(x+20, y, x+40, y-40);
}
```

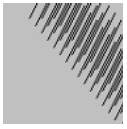
0-05

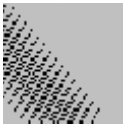
A function is a set of code within a program that performs a specific task. Functions are powerful programming tools that make programs easier to read and change. The `diagonals()` function in the following example was written to draw a sequence of three diagonal lines each time it is run inside `draw()`. Two *parameters*, the numbers in the parentheses after the function name, set the position of the lines. These numbers are passed into the function definition on line 12 and are used as the values for the variables *x* and *y* in lines 13–15. Functions are discussed in more depth on pages 181–196.

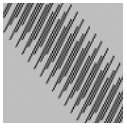
0-06

```
void setup() {
  size(100, 100);
  noLoop();
}

void draw() {
  diagonals(40, 90);
  diagonals(60, 62);
  diagonals(20, 40);
}

void diagonals(int x, int y) {
  line(x, y, x+20, y-40);
  line(x+10, y, x+30, y-40);
  line(x+20, y, x+40, y-40);
}
```

The variables used in the previous programs each store one data element. If we want to have 20 groups of lines on screen, it will require 40 variables: 20 for the horizontal positions and 20 for the vertical positions. This can make programming tedious and can make programs difficult to read. Instead of using multiple variable names, we can use *arrays*. An array can store a list of data elements as a single name. A `for` structure can be used to cycle through each array element in sequence. Arrays are discussed on pages 301–313, and the `for` structure is discussed on pages 61–68.







0-07

```
int num = 20;
int[] dx = new int[num];  // Declare and create an array
int[] dy = new int[num];  // Declare and create an array

void setup() {
  size(100, 100);
  for (int i = 0; i < num; i++) {
    dx[i] = i * 5;
    dy[i] = 12 + (i * 6);
  }
}

void draw() {
  background(204);
  for (int i = 0; i < num; i++) {
    dx[i] = dx[i] + 1;
    if (dx[i] > 100) {
      dx[i] = -100;
    }
```

```
    diagonals(dx[i], dy[i]);
  }
}

void diagonals(int x, int y) {
  line(x, y, x+20, y-40);
  line(x+10, y, x+30, y-40);
  line(x+20, y, x+40, y-40);
}
```

0-07 cont.

Object-oriented programming is a way of structuring code into *objects*, units of code that contain both data and functions. This style of programming makes a strong connection between groups of data and the functions that act on this data. The `diagonals()` function can be expanded by making it part of a *class* definition. Objects are created using the class as a template. The variables for positioning the lines and setting their drawing attributes then move inside the class definition to be more closely associated with drawing the lines. Object-oriented programming is discussed further on pages 395–411.

0-08

```
Diagonals da, db;

void setup() {
  size(100, 100);
  smooth();
  // Inputs: x, y, speed, thick, gray
  da = new Diagonals(0, 80, 1, 2, 0);
  db = new Diagonals(0, 55, 2, 6, 255);
}

void draw() {
  background(204);
  da.update();
  db.update();
}

class Diagonals {
  int x, y, speed, thick, gray;

  Diagonals(int xpos, int ypos, int s, int t, int g) {
    x = xpos;
    y = ypos;
    speed = s;
    thick = t;
    gray = g;
  }
```

0-08
cont.

```
  void update() {
    strokeWeight(thick);
    stroke(gray);
    line(x, y, x+20, y-40);
    line(x+10, y, x+30, y-40);
    line(x+20, y, x+40, y-40);
    x = x + speed;
    if (x > 100) {
      x = -100;
    }
  }
}
```

This short walk-through serves to introduce, but not fully explain, some of the core concepts explored in this text. Many key ideas of working with software were mentioned only briefly and others were omitted. Each topic is covered in depth later in the book.

Reference

The reference for the Processing language complements the text in this book. We advise keeping the reference open and consulting it while programming. The reference can be accessed by selecting the "Reference" option from the Help menu within Processing. It's also available online at *www.processing.org/reference*. The reference can also be accessed within the text window. Highlight a word, right-click (or Ctrl-click in Mac OS X), and select "Find in Reference" from the menu that appears. You can also select "Find in Reference" from the Help menu. There are two versions of the Processing reference. The Abridged Reference lists the elements of the Processing language introduced in this book, and the Complete Reference documents additional features.

Structure 1: Code Elements

This unit introduces the most basic elements and vocabulary for writing software.

Syntax introduced:
`// (comment), /* */ (multiline comment)`
`“;” (statement terminator), “,” (comma)`
`print(), println()`

Creating software is an act of *writing*. Before starting to write code, it’s important to acknowledge the difference between writing a computer program and writing an Email or an essay. Writing in a human language allows the author to utilize the ambiguity of words and to have great flexibility in constructing phrases. These techniques allow multiple interpretations of a single text and give each author a unique voice. Each computer program also reveals the style of its author, but there is far less room for ambiguity. While people can interpret vague meanings and can usually disregard poor grammar, computers cannot. Some of the linguistic details of writing code are discussed here to prevent early frustration. If you keep these details in mind as you begin to program, they will gradually become habitual. This unit presents variations of a simple program that sets the size and background color of the display window, demonstrating some of the most basic elements of writing code with Processing.

Comments

Comments are ignored by the computer but are important for people. They let you write notes to yourself and to others who read your programs. Because programs use symbols and arcane notation to describe complex procedures, it is often difficult to remember how individual parts of a program work. Good comments serve as reminders when you revisit a program and explain your thoughts to others reading the code. Commented sections appear in a different color than the rest of the code. This program explains how comments work:

1-01

```
// Two forward slashes are used to denote a comment.
// All text on the same line is a part of the comment.
// There must be no spaces between the slashes. For example,
// the code "/ /" is not a comment and will cause an error

// If you want to have a comment that is many
// lines long, you may prefer to use the syntax for a
// multiline comment
```

```
/*                                                          1-01
  A forward slash followed by an asterisk allows the        cont.
  comment to continue until the opposite
*/

// All letters and symbols that are not comments are translated
// by the compiler. Because the following lines are not comments,
// they are run and draw a display window of 200 x 200 pixels
size(200, 200);
background(102);
```

Functions

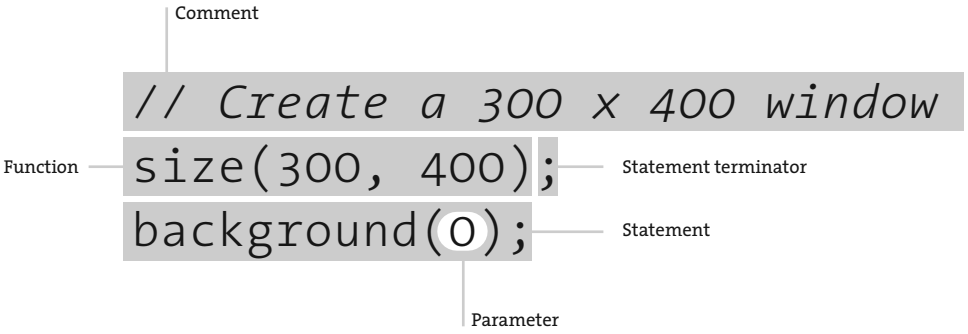
Functions allow you to draw shapes, set colors, calculate numbers, and to execute many other types of actions. A function's name is usually a lowercase word followed by parentheses. The comma-separated elements between the parentheses are called parameters, and they affect the way the function works. Some functions have no parameters and others have many. This program demonstrates the `size()` and `background()` functions.

```
// The size function has two parameters. The first sets the width      1-02
// of the display window and the second sets the height
size(200, 200);

// This version of the background function has one parameter.
// It sets the gray value for the background of the display window
// in the range of 0 (black) to 255 (white)
background(102);
```

Expressions, Statements

Using an analogy to human languages, a software expression is like a phrase. Software expressions are often combinations of operators such as +, *, and / that operate on the values to their left and right. A software expression can be as basic as a single number or can be a long combination of elements. An expression always has a value, determined by evaluating its contents.

Expression	*Value*
`5`	`5`
`122.3+3.1`	`125.4`
`((3+2)*-10) + 1`	`-49`



Anatomy of a program
Every program is composed of different language elements. These elements work together to describe the intentions of the programmer so they can be interpreted by a computer. The anatomy of a more complicated program is shown on page 176.

Expressions can also compare two values with operators such as > (greater than) and < (less than). These comparisons are evaluated as `true` or `false`.

Expression	*Value*
`6 > 3`	`true`
`54 < 50`	`false`

A set of expressions together create a statement, the programming equivalent of a sentence. It's a complete unit that ends with the statement terminator, the programming equivalent of a period. In the Processing language, the statement terminator is a semicolon.

Just as there are different types of sentences, there are different types of statements. A statement can define a variable, assign a variable, run a function, or construct an object. Each will be explained in more detail later, but examples are shown here:

```
size(200, 200);   // Runs the size() function                    1-03
int x;            // Declares a new variable x
x = 102;          // Assigns the value 102 to the variable x
background(x);    // Runs the background() function
```

Omitting the semicolon at the end of a statement, a very common mistake, will result in an error message, and the program will not run.

Case sensitivity

In written English, some words are capitalized and others are not. Proper nouns like Ohio and John and the first letter of every sentence are capitalized, while most other words are lowercase. In many programming languages, some parts of the language must be capitalized and others must be lowercase. Processing differentiates between uppercase and lowercase characters; therefore, writing "Size" when you mean to write "size" creates an error. You must be exacting in adhering to the capitalization rules.

1-04
```
size(200, 200);
Background(102);  // ERROR! The B in "background" is capitalized
```

Whitespace

In many programming languages, including Processing, there can be an arbitrary amount of space between the elements of a program. Unlike the rigorous syntax of statement terminators, spacing does not matter. The following two lines of code are a standard way of writing a program:

1-05
```
size(200, 200);
background(102);
```

However, the whitespace between the code elements can be set to any amount and the program will run exactly the same way:

1-06
```
size

(   200,
  200)             ;
background   (          102)
        ;
```

Console

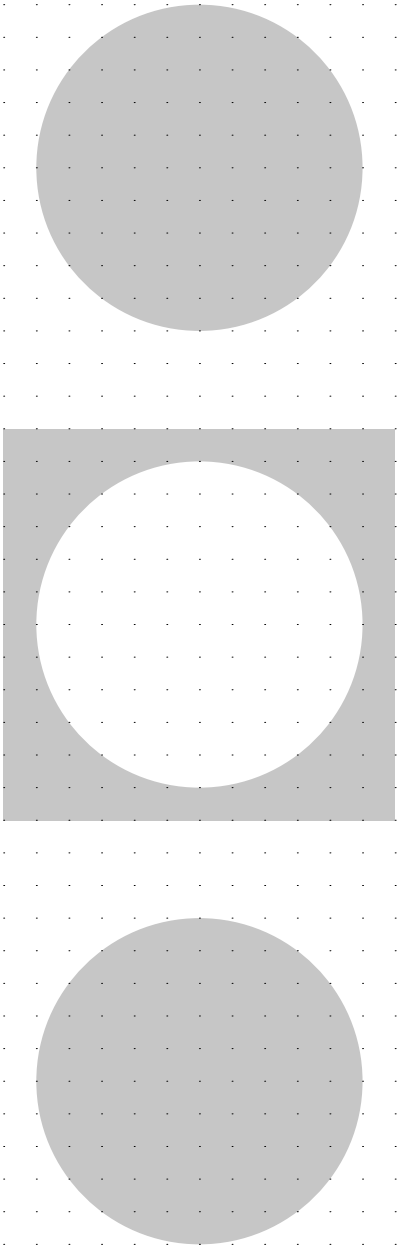
When software runs, the computer performs operations at a rate too fast to perceive with human eyes. Because it is important to understand what is happening inside the machine, the functions `print()` and `println()` can be used to display data while a program is running. These functions don't send pages to a printer, but instead write text to the console (pp. 8, 9). The console can be used to display a variable, confirm an event, or check incoming data from an external device. Such uses might not seem clear now, but they will reveal themselves over the course of this book. Like comments, `print()` and `println()` can clarify the intentions and execution of computer programs.

1-07

```
// To print text to the screen, place the desired output in quotes
println("Processing...");  // Prints "Processing..." to the console

// To print the value of a variable, rather than its name,
// don't put the name of the variable in quotes
int x = 20;
println(x);  // Prints "20" to the console

// While println() moves to the next line after the text
// is output, print() does not
print("10");
println("20");  // Prints "1020" to the console
println("30");  // Prints "30" to the console

// The "+" operator can be used for combining multiple text
// elements into one line
int x2 = 20;
int y2 = 80;
println(x2 + " : " + y2);  // Prints "20 : 80" to the message window
```

Exercises

1. *Write comments in the text area explaining a piece of software you would like to write.*
2. *Write a program to make a 640 × 480 pixel display window with a black background.*
3. *Use `print()` and `println()` to write some text to the console.*



Shape 1: Coordinates, Primitives

This unit introduces the coordinate system of the display window and a variety of geometric shapes.

Syntax introduced:

```
size(), point(), line(), triangle(), quad(), rect(), ellipse(), bezier()
background(), fill(), stroke(), noFill(), noStroke()
strokeWeight(), strokeCap(), strokeJoin()
smooth(), noSmooth(), ellipseMode(), rectMode()
```

Drawing a shape with code can be difficult because every aspect of its location must be specified with a number. When you're accustomed to drawing with a pencil or moving shapes around on a screen with a mouse, it can take time to start thinking in relation to the screen's strict coordinate grid. The mental gap between seeing a composition on paper or in your mind and translating it into code notation is wide, but easily bridged.

Coordinates

Before making a drawing, it's important to think about the dimensions and qualities of the surface to which you'll be drawing. If you're making a drawing on paper, you can choose from myriad utensils and papers. For quick sketching, newsprint and charcoal are appropriate. For a refined drawing, a smooth handmade paper and range of pencils may be preferred. In contrast, when you are drawing to a computer's screen, the primary options available are the size of the window and the background color.

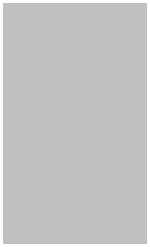
A computer screen is a grid of small light elements called pixels. Screens come in many sizes and resolutions. We have three different types of computer screens in our studios, and they all have a different number of pixels. The laptops have 1,764,000 pixels (1680 wide × 1050 high), the flat panels have 1,310,720 pixels (1280 wide × 1024 high), and the older monitors have 786,432 pixels (1024 wide × 768 high). Millions of pixels may sound like a vast quantity, but they produce a poor visual resolution compared to physical media such as paper. Contemporary screens have a resolution around 100 dots per inch, while many modern printers provide more than 1000 dots per inch. On the other hand, paper images are fixed, but screens have the advantage of being able to change their image many times per second.

Processing programs can control all or a subset of the screen's pixels. When you click the Run button, a display window opens and allows access to reading and writing the pixels within. It's possible to create images larger than the screen, but in most cases you'll make a window the size of the screen or smaller.

The size of the display window is controlled with the `size()` function:

```
size(width, height)
```

The `size()` function has two parameters: the first sets the width of the window and the second sets its height.

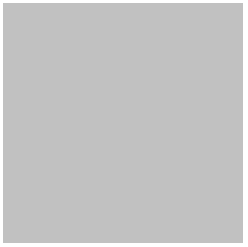


```
// Draw the display window 120 pixels
// wide and 200 pixels high
size(120, 200);
```

2-01

```
// Draw the display window 320 pixels
// wide and 240 pixels high
size(320, 240);
```

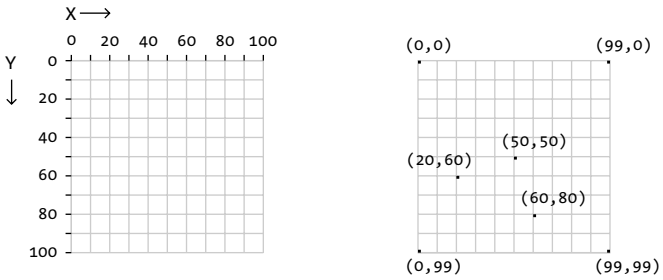
2-02



```
// Draw the display window 200 pixels
// wide and 200 pixels high
size(200, 200);
```

2-03

A position on the screen is comprised of an x-coordinate and a y-coordinate. The x-coordinate is the horizontal distance from the origin and the y-coordinate is the vertical distance. In Processing, the origin is the upper-left corner of the display window and coordinate values increase down and to the right. The image on the left shows the coordinate system, and the image on the right shows a few coordinates placed on the grid:



A position is written as the x-coordinate value followed by the y-coordinate, separated with a comma. The notation for the origin is (0,0), the coordinate (50,50) has an x-coordinate of 50 and a y-coordinate of 50, and the coordinate (20,60) is an x-coordinate of 20 and a y-coordinate of 60. If the size of the display window is 100 pixels wide and 100 pixels high, (0,0) is the pixel in the upper-left corner, (99,0) is the pixel in the upper-right corner, (0,99) is the pixel in the lower-left corner, and (99,99) is the pixel in the lower-right corner. This becomes clearer when we look at examples using `point()`.

Primitive shapes

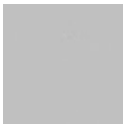
A point is the simplest visual element and is drawn with the `point()` function:

```
point(x, y)
```

This function has two parameters: the first is the x-coordinate and the second is the y-coordinate. Unless specified otherwise, a point is the size of a single pixel.

2-04

```
// Points with the same X and Y parameters
// form a diagonal line from the
// upper-left corner to the lower-right corner
point(20, 20);
point(30, 30);
point(40, 40);
point(50, 50);
point(60, 60);
```



```
// Points with the same Y parameter have the
// same distance from the top and bottom
// edges of the frame
point(50, 30);
point(55, 30);
point(60, 30);
point(65, 30);
point(70, 30);
```

2-05

```
// Points with the same X parameter have the
// same distance from the left and right
// edges of the frame
point(70, 50);
point(70, 55);
point(70, 60);
point(70, 65);
point(70, 70);
```

2-06

```
// Placing a group of points next to one
// another creates a line
point(50, 50);
point(50, 51);
point(50, 52);
point(50, 53);
point(50, 54);
point(50, 55);
point(50, 56);
point(50, 57);
point(50, 58);
point(50, 59);
```

2-07

```
// Setting points outside the display
// area will not cause an error,
// but the points won't be visible
point(-500, 100);
point(400, -600);
point(140, 2500);
point(2500, 100);
```

2-08

While it's possible to draw any line as a series of points, lines are more simply drawn with the `line()` function. This function has four parameters, two for each endpoint:

```
line(x1, y1, x2, y2)
```

The first two parameters set the position where the line starts and the last two set the position where the line stops.

```
// When the y-coordinates for a line are the
// same, the line is horizontal
line(10, 30, 90, 30);
line(10, 40, 90, 40);
line(10, 50, 90, 50);
```

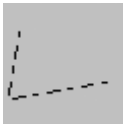
2-09

```
// When the x-coordinates for a line are the
// same, the line is vertical
line(40, 10, 40, 90);
line(50, 10, 50, 90);
line(60, 10, 60, 90);
```

2-10

```
// When all four parameters are different,
// the lines are diagonal
line(25, 90, 80, 60);
line(50, 12, 42, 90);
line(45, 30, 18, 36);
```

2-11



```
// When two lines share the same point they connect
line(15, 20, 5, 80);
line(90, 65, 5, 80);
```

2-12

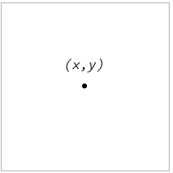
The `triangle()` function draws triangles. It has six parameters, two for each point:

```
triangle(x1, y1, x2, y2, x3, y3)
```

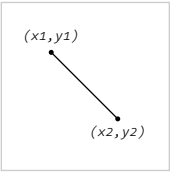
The first pair defines the first point, the middle pair the second point, and the last pair the third point. Any triangle can be drawn by connecting three lines, but the `triangle()` function makes it possible to draw a filled shape. Triangles of all shapes and sizes can be created by changing the parameter values.

```
triangle(60, 10, 25, 60, 75, 65);  // Filled triangle
line(60, 30, 25, 80);  // Outlined triangle edge
line(25, 80, 75, 85);  // Outlined triangle edge
line(75, 85, 60, 30);  // Outlined triangle edge
```

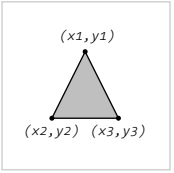
2-13



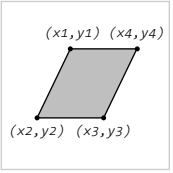
point(x, y)



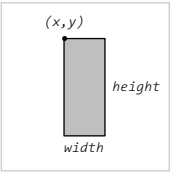
line(x1, y1, x2, y2)



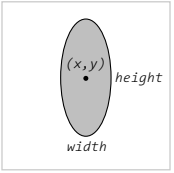
triangle(x1, y1, x2, y2, x3, y3)



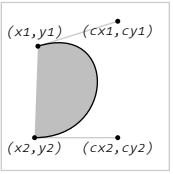
quad(x1, y1, x2, y2, x3, y3, x4, y4)



rect(x, y, width, height)



ellipse(x, y, width, height)



bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)

Geometry primitives

Processing has seven functions to assist in making simple shapes. These images show the format for each. Replace the parameters with numbers to use them within a program. These functions are demonstrated in codes 2-04 to 2-22.

```
triangle(55, 9, 110, 100, 85, 100);
triangle(55, 9, 85, 100, 75, 100);
triangle(-1, 46, 16, 34, -7, 100);
triangle(16, 34, -7, 100, 40, 100);
```

2-14

The `quad()` function draws a quadrilateral, a four-sided polygon. The function has eight parameters, two for each point.

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

Changing the parameter values can yield rectangles, squares, parallelograms, and irregular quadrilaterals.

```
quad(38, 31, 86, 20, 69, 63, 30, 76);
```

2-15

```
quad(20, 20, 20, 70, 60, 90, 60, 40);
quad(20, 20, 70, -20, 110, 0, 60, 40);
```

2-16

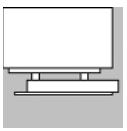
Drawing rectangles and ellipses works differently than the shapes previously introduced. Instead of defining each point, the four parameters set the position and the dimensions of the shape. The `rect()` function draws a rectangle:

```
rect(x, y, width, height)
```

The first two parameters set the location of the upper-left corner, the third sets the width, and the fourth sets the height. Use the same value for the *`width`* and *`height`* parameters to draw a square.

```
rect(15, 15, 40, 40);  // Large square
rect(55, 55, 25, 25);  // Small square
```

2-17



```
rect(0, 0, 90, 50);
rect(5, 50, 75, 4);
rect(24, 54, 6, 6);
rect(64, 54, 6, 6);
rect(20, 60, 75, 10);
rect(10, 70, 80, 2);
```

2-18

The `ellipse()` function draws an ellipse in the display window:

```
ellipse(x, y, width, height)
```

The first two parameters set the location of the center of the ellipse, the third sets the width, and the fourth sets the height. Use the same value for the *`width`* and *`height`* parameters to draw a circle.

```
ellipse(40, 40, 60, 60);  // Large circle
ellipse(75, 75, 32, 32);  // Small circle
```

2-19

```
ellipse(35, 0, 120, 120);
ellipse(38, 62, 6, 6);
ellipse(40, 100, 70, 70);
```

2-20

The `bezier()` function can draw lines that are not straight. A Bézier curve is defined by a series of control points and anchor points. A curve is drawn between the anchor points, and the control points define its shape:

```
bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)
```

The function requires eight parameters to set four points. The curve is drawn between the first and fourth points, and the control points are defined by the second and third points. In software that uses Bézier curves, such as Adobe Illustrator, the control points are represented by the tiny handles that protrude from the edge of a curve.

```
bezier(32, 20, 80, 5, 80, 75, 30, 75);
// Draw the control points
line(32, 20, 80, 5);
ellipse(80, 5, 4, 4);
line(80, 75, 30, 75);
ellipse(80, 75, 4, 4);
```

2-21

```
bezier(85, 20, 40, 10, 60, 90, 15, 80);
// Draw the control points
line(85, 20, 40, 10);
ellipse(40, 10, 4, 4);
line(60, 90, 15, 80);
ellipse(60, 90, 4, 4);
```

2-22

Drawing order

The order in which shapes are drawn in the code defines which shapes appear on top of others in the display window. If a rectangle is drawn in the first line of a program, it is drawn behind an ellipse drawn in the second line of the program. Reversing the order places the rectangle on top.

```
rect(15, 15, 50, 50);     // Bottom
ellipse(60, 60, 55, 55);  // Top
```

2-23

```
ellipse(60, 60, 55, 55);  // Bottom
rect(15, 15, 50, 50);     // Top
```

2-24

Gray values

The examples so far have used the default light-gray background, black lines, and white shapes. To change these default values, it's necessary to introduce additional syntax. The `background()` function sets the color of the display window with a number between 0 and 255. This range may be awkward if you're not familiar with drawing software on the computer. The value 255 is white and the value 0 is black, with a range of gray values in between. If no background value is defined, the default value 204 (light gray) is used.

```
background(0);
```

2-25

```
background(124);
```

2-26


```
background(230);
```

2-27

The `fill()` function sets the fill value of shapes, and the `stroke()` function sets the outline value of the drawn shapes. If no fill value is defined, the default value of 255 (white) is used. If no stroke value is defined, the default value of 0 (black) is used.

2-28

```
rect(10, 10, 50, 50);
fill(204);  // Light gray
rect(20, 20, 50, 50);
fill(153);  // Middle gray
rect(30, 30, 50, 50);
fill(102);  // Dark gray
rect(40, 40, 50, 50);
```

2-29

```
background(0);
rect(10, 10, 50, 50);
stroke(102); // Dark gray
rect(20, 20, 50, 50);
stroke(153); // Middle gray
rect(30, 30, 50, 50);
stroke(204); // Light gray
rect(40, 40, 50, 50);
```

Once a fill or stroke value is defined, it applies to all shapes drawn afterward. To change the fill or stroke value, use the `fill()` or `stroke()` function again.

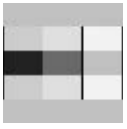
2-30

```
fill(255);  // White
rect(10, 10, 50, 50);
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
fill(0);  // Black
rect(40, 40, 50, 50);
```

An optional second parameter to `fill()` and `stroke()` controls transparency. Setting the parameter to 255 makes the shape entirely opaque, and 0 is totally transparent:

2-31

```
background(0);
fill(255, 220);
rect(15, 15, 50, 50);
rect(35, 35, 50, 50);
```



2-32

```
fill(0);
rect(0, 40, 100, 20);
fill(255, 51);   // Low opacity
rect(0, 20, 33, 60);
fill(255, 127);  // Medium opacity
```

```
rect(33, 20, 33, 60);
fill(255, 204);  // High opacity
rect(66, 20, 33, 60);
```

2-32 cont.

The stroke and fill of a shape can be disabled. The `noFill()` function stops Processing from filling shapes, and the `noStroke()` function stops lines from being drawn and shapes from having outlines. If `noFill()` and `noStroke()` are both used, nothing will be drawn to the screen.

```
rect(10, 10, 50, 50);
noFill();  // Disable the fill
rect(20, 20, 50, 50);
rect(30, 30, 50, 50);
```

2-33

```
rect(20, 15, 20, 70);
noStroke();  // Disable the stroke
rect(50, 15, 20, 70);
rect(80, 15, 20, 70);
```

2-34

Setting color fill and stroke values is introduced in Color 1 (p. 85).

Drawing attributes

In addition to changing the fill and stroke values of shapes, it's also possible to change attributes of the geometry. The `smooth()` and `noSmooth()` functions enable and disable smoothing (also called antialiasing). Once these functions are used, all shapes drawn afterward are affected. If `smooth()` is used first, using `noSmooth()` cancels the setting, and vice versa.

```
ellipse(30, 48, 36, 36);
smooth();
ellipse(70, 48, 36, 36);
```

2-35

```
smooth();
ellipse(30, 48, 36, 36);
noSmooth();
ellipse(70, 48, 36, 36);
```

2-36

Line attributes are controlled by the `strokeWeight()`, `strokeCap()`, and `strokeJoin()` functions. The `strokeWeight()` function has one numeric parameter that sets the thickness of all lines drawn after the function is used. The `strokeCap()` function requires one parameter that can be either `ROUND`, `SQUARE`, or `PROJECT`.

ROUND makes round endpoints, and SQUARE squares them. PROJECT is a mix of the two that extends a SQUARE endpoint by the radius of the line. The strokeJoin() function has one parameter that can be either BEVEL, MITER, or ROUND. These parameters determine the way line segments or the stroke around a shape connects. BEVEL causes lines to join with squared corners, MITER is the default and joins lines with pointed corners, and ROUND creates a curve.

2-37

```
smooth();
line(20, 20, 80, 20);  // Default line weight of 1
strokeWeight(6);
line(20, 40, 80, 40);  // Thicker line
strokeWeight(18);
line(20, 70, 80, 70);  // Beastly line
```

2-38

```
smooth();
strokeWeight(12);
strokeCap(ROUND);
line(20, 30, 80, 30);  // Top line
strokeCap(SQUARE);
line(20, 50, 80, 50);  // Middle line
strokeCap(PROJECT);
line(20, 70, 80, 70);  // Bottom line
```

2-39

```
smooth();
strokeWeight(12);
strokeJoin(BEVEL);
rect(12, 33, 15, 33);  // Left shape
strokeJoin(MITER);
rect(42, 33, 15, 33);  // Middle shape
strokeJoin(ROUND);
rect(72, 33, 15, 33);  // Right shape
```

Shape 2 (p. 69) and Shape 3 (p. 197) show how to draw shapes with more flexibility.

Drawing modes

By default, the parameters for ellipse() set the x-coordinate of the center, the y-coordinate of the center, the width, and the height. The ellipseMode() function changes the way these parameters are used to draw ellipses. The ellipseMode() function requires one parameter that can be either CENTER, RADIUS, CORNER, or CORNERS. The default mode is CENTER. The RADIUS mode also uses the first and second parameters of ellipse() to set the center, but causes the third parameter to set half of

the width and the fourth parameter to set half of the height. The CORNER mode makes ellipse() work similarly to rect(). It causes the first and second parameters to position the upper-left corner of the rectangle that circumscribes the ellipse and uses the third and fourth parameters to set the width and height. The CORNERS mode has a similar affect to CORNER, but is causes the third and fourth parameters to ellipse() to set the lower-right corner of the rectangle.

2-40

```
smooth();
noStroke();
ellipseMode(RADIUS);
fill(126);
ellipse(33, 33, 60, 60);  // Gray ellipse
fill(255);
ellipseMode(CORNER);
ellipse(33, 33, 60, 60);  // White ellipse
fill(0);
ellipseMode(CORNERS);
ellipse(33, 33, 60, 60);  // Black ellipse
```

In a similar fashion, the rectMode() function affects how rectangles are drawn. It requires one parameter that can be either CORNER, CORNERS, or CENTER. The default mode is CORNER, and CORNERS causes the third and fourth parameters of rect() to draw the corner opposite the first. The CENTER mode causes the first and second parameters of rect() to set the center of the rectangle and uses the third and fourth parameters as the width and height.

2-41

```
noStroke();
rectMode(CORNER);
fill(126);
rect(40, 40, 60, 60);     // Gray ellipse
rectMode(CENTER);
fill(255);
rect(40, 40, 60, 60);     // White ellipse
rectMode(CORNERS);
fill(0);
rect(40, 40, 60, 60);     // Black ellipse
```

Exercises

1. *Create a composition by carefully positioning one line and one ellipse.*
2. *Modify the code for exercise 1 to change the fill, stroke, and background values.*
3. *Create a visual knot using only Bézier curves.*

false	84	6	456749	590203.000
true	12	*	418953	1209181.500
true	-64	D	485484	2082378.000
false	96	"	1895150	-1391668.750
true	-48	,	-567039	-1517834.000
false	-123	x	200745	-2107862.750
false	15	)	-1061358	537499.250
true	-23	!	1186751	2047401.000
false	47	M	1579167	1057598.500
false	59	U	1362780	-2092479.125
false	-23	Q	969695	1715130.000
false	-94	S	-1790733	-1348664.000
false	-50	c	734766	923550.750
true	-92	A	-159095	-969601.250
true	42	&	813913	-651030.125
false	-41	Y	1543424	1857006.000
false	62	q	145359	483532.000
true	-83	=	-1213025	-1273871.000
false	-107	%	-366036	825891.750
true	69	y	913485	815780.500
true	-116	i	-450082	1912691.000
false	-125	Z	-454392	479536.000
false	-122	i	1846579	2119445.000
true	97	M	-1416086	-1530103.000
false	-80	d	1736539	761730.000
true	77	]	-251570	-770656.500
true	-104	p	-122459	7880.500
false	125	U	2015498	281250.000
true	-32	$	1235768	-1839862.000
true	-121	*	-70136	-2110472.000
false	-5	|	-644916	1148654.500
false	0	\	-1050010	-697901.375
true	61	o	973362	-1923781.750
true	-126	2	-627696	1375153.750
true	108	o	1440987	-2131691.000
false	-49	c	135757	-848097.250
false	-48	:	1884981	1607443.250
false	27	S	1755091	-1509226.500
true	122	\	-771512	-13727.750
true	1	N	88857	-1286271.875
true	-31	L	225379	876042.750
true	-46	9	-2097426	1962124.500
false	66	f	1030910	460799.250
true	-68	U	-476555	-1738484.250
false	-42	/	347440	-2130674.500
false	-60	%	1687135	1093877.250
true	-72	H	-548337	805527.750
true	-24	_	-631134	-1417360.000
false	-39	e	1491429	1694085.750
false	20	q	46194	-1970949.250
true	37	R	486794	2031981.250
true	28	F	-2030792	1623354.000
false	44	[	-855542	1365101.250
false	61	=	564487	-372181.625

Data 1: Variables

This unit introduces different types of data and explains how to create variables and assign them values.

Syntax introduced:
`int, float, boolean, true, false, = (assign), width, height`

What is data? Data often consists of measurements of physical characteristics. For example, Casey's California driver's license states his sex is M, his hair is BRN, and his eyes are HZL. The values M, BRN, and HZL are items of data associated with Casey. Data can be the population of a country, the average annual rainfall in Los Angeles, or your current heart rate. In software, data is stored as numbers and characters. Examples of digital data include a photograph of a friend stored on your hard drive, a song downloaded from the Internet, and a news article loaded through a web browser. Less obvious is the data continually created and exchanged between computers and other devices. For example, computers are continually receiving data from the mouse and keyboard. When writing a program, you might create a data element to save the location of a shape, to store a color for later use, or to continuously measure changes in cursor position.

Data types

Processing can store and modify many different kinds of data, including numbers, letters, words, colors, images, fonts, and boolean values (`true`, `false`). The computer stores each in a different way, so it has to know which type of data is being used to know how to manage it. For example, storing a word takes more room than storing one letter; therefore, storing the word *Cincinnati* requires more space than storing the letter *C*. If space has been allocated for only one letter, trying to store a word in the same space will cause an error. Every data element is represented as sequences of bits (0s and 1s) in the computer's memory (more information about bits is found in Appendix D, p. 669). For example, 01000001 can be interpreted as the letter *A*, and it can also be interpreted as the number 65. It's necessary to specify the type of data so the computer knows how to correctly interpret the bits.

Numeric data is the first type of data encountered in the following sections of this book. There are two types of numeric data used in Processing: integer and floating-point. Integers are whole numbers such as 12, –120, 8, and 934. Processing represents integer data with the `int` data type. Floating-point numbers have a decimal point for creating fractions of whole numbers such as 12.8, –120.75, 8.125, and 934.82736. Processing represents floating-point data with the `float` data type. Floating-point numbers are often used to approximate analog or continuous values because they have decimal

resolution. For example, using integer values, there is only one number between 3 and 5, but floating-point numbers allow us to express myriad numbers between such as 4.0, 4.5, 4.75, 4.825, etc. Both int and float values may be positive, negative, or zero.

The simplest data element in Processing is a boolean variable. Variables of this type can have only one of two values—`true` or `false`. The name boolean refers to the mathematician George Boole (b. 1815), the inventor of Boolean algebra—the foundation for how digital computers work. A boolean variable is often used to make decisions about which lines of code are run and which are ignored.

The following table compares the capacities of the data types mentioned above with other common data types:

Name	Size	Value range
boolean	1 bit	true or false
byte	8 bits	-128 to 127
char	16 bits	0 to 65535
int	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.40282347E+38 to -3.40282347E+38
color	32 bits	16,777,216 colors

Additional types of data are introduced and explained in Data 2 (p. 101), Data 3 (p. 105), Image 1 (p. 95), Typography 1 (p. 111), and Structure 4 (p. 395).

Variables

A variable is a container for storing data. Variables allow a data element to be reused many times within a program. Every variable has two parts, a name and a value. If the number 21 is stored in the variable called *age*, every time the word *age* appears in the program, it will be replaced with the value 21 when the code is run. In addition to its name and value, every variable has a data type that defines the category of data it can hold.

A variable must be declared before it is used. A variable declaration states the data type and variable name. The following lines declare variables and then assign values to the variables:

3-01

```
int x;      // Declare the variable x of type int
float y;    // Declare the variable y of type float
boolean b;  // Declare the variable b of type boolean
x = 50;     // Assign the value 50 to x
y = 12.6;   // Assign the value 12.6 to y
b = true;   // Assign the value true to b
```

As a shortcut, a variable can be declared and assigned on the same line:

```
int x = 50;                                                    3-02
float y = 12.6;
boolean b = true;
```

More than one variable can be declared in one line, and the variables can then be assigned separately:

```
float x, y, z;                                                 3-03
x = -3.9;
y = 10.1;
z = 124.23;
```

When a variable is declared, it is necessary to state the data type before its name; but after it's declared, the data type cannot be changed or restated. If the data type is included again for the same variable, the computer will interpret this as an attempt to make a new variable with the same name, and this will cause an error (an exception to this rule is made when each variable has a different scope, p. 178):

```
int x = 69;  // Assign 69 to x                                 3-04
x = 70;      // Assign 70 to x
int x = 71;  // ERROR! The data type for x is duplicated
```

The = symbol is called the assignment operator. It assigns the value from the right side of the = to the variable on its left. Values can be assigned only to variables. Trying to assign a constant to another constant produces an error:

```
// Error! The left side of an assignment must be a variable    3-05
5 = 12;
```

When working with variables of different types in the same program, be careful not to mix types in a way that causes an error. For example, it's not possible to fit a floating-point number into an integer variable:

```
// Error! It's not possible to fit a floating-point number into an int  3-06
int x = 24.8;
```

```
float f = 12.5;                                                3-07
// Error! It's not possible to fit a floating-point number into an int
int y = f;
```

Variables should have names that describe their content. This makes programs easier to read and can reduce the need for verbose commenting. It's up to the programmer to decide how she will name variables. For example, a variable storing the room temperature could logically have the following names:

```
t
temp
temperature
roomTemp
roomTemperature
```

Variables like `t` should be used minimally or not at all because they are cryptic—there's no hint as to what they contain. However, long names such as *roomTemperature* can also make code tedious to read. If we were writing a program with this variable, our preference might be to use the name *roomTemp* because it is both concise and descriptive. The name *temp* could also work, but because it's used commonly as an abbreviation for "temporary," it wouldn't be the best choice.

There are a few conventions that make it easier for other people to read your programs. Variables' names should start with a lowercase letter, and if there are multiple words in the name, the first letter of each additional word should be capitalized. There are a few absolute rules in naming variables. Variable names cannot start with numbers, and they must not be a reserved word. Examples of reserved words include `int`, `if`, `true`, and `null`. A complete list is found in Appendix B (p. 663). To avoid confusion, variables should not have the same names as elements of the Processing language such as `line` and `ellipse`. The complete Processing language is listed in the reference included with the software.

Another important consideration related to variables is the scope (p. 178). The scope of a variable defines where it can be used relative to where it's created.

Processing variables

The Processing language has built-in variables for storing commonly used data. The width and height of the display window are stored in variables called `width` and `height`. If a program doesn't include `size()`, the `width` and `height` variables are both set to 100. Test by running the following programs

```
println(width + ", " + height);  // Prints "100, 100" to the console
```
3-08

```
size(300, 400);
println(width + ", " + height);  // Prints "300, 400" to the console
```
3-09

```
size(1280, 1024);
println(width + ", " + height);  // Prints "1280, 1024" to the console
```
3-10

Using the `width` and `height` variables is useful when writing a program to scale to different sizes. This technique allows a simple change to the parameters of `size()` to alter the dimensions and proportions of a program, rather than changing values throughout the code. Run the following code with different values in the `size()` function to see it scale to every window size.

3-11

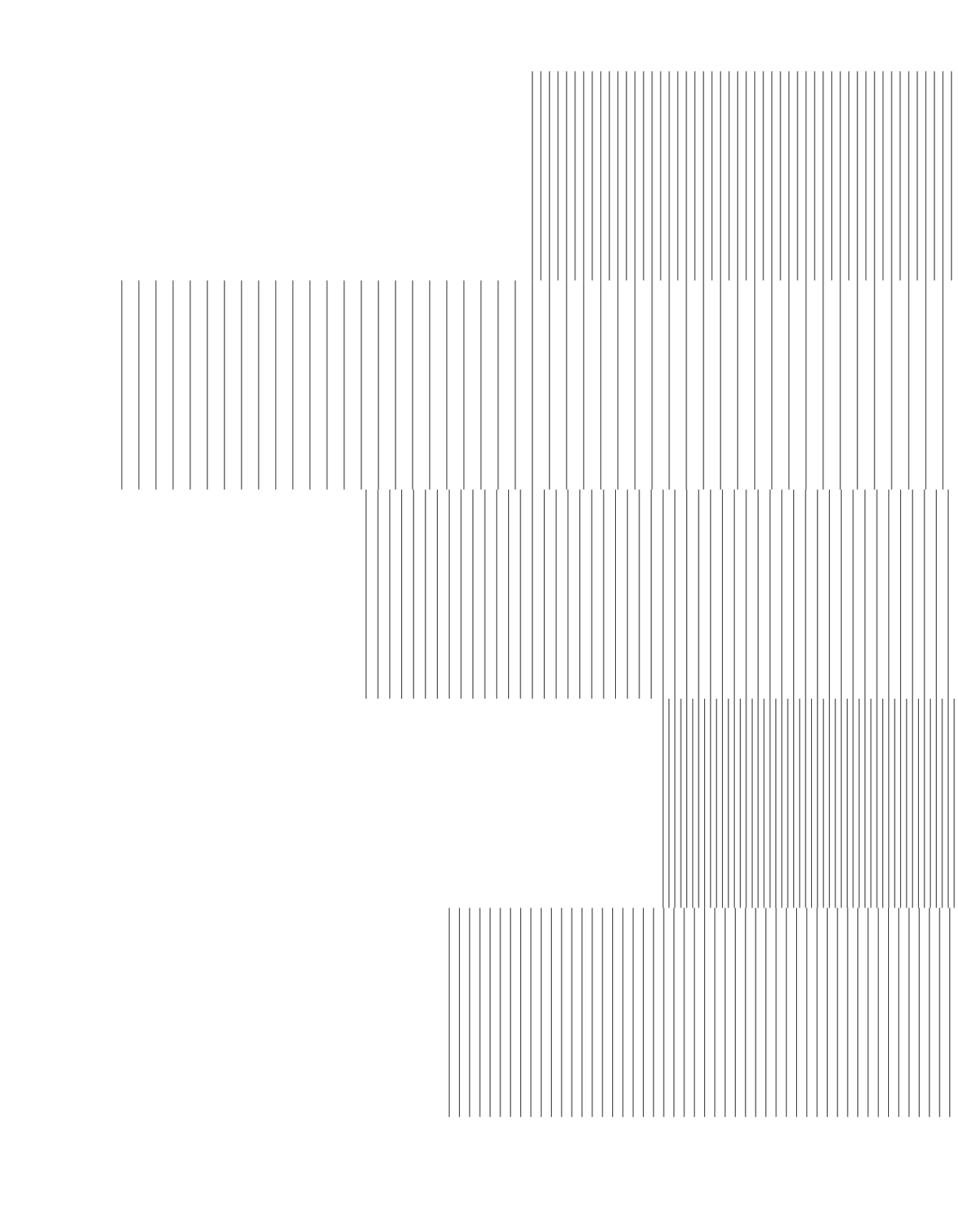
```
size(100, 100);
ellipse(width*0.5, height*0.5, width*0.66, height*0.66);
line(width*0.5, 0, width*0.5, height);
line(0, height*0.5, width, height*0.5);
```

You should always use actual numbers in `size()` instead of variables. When a sketch is exported, these numbers are used to determine the dimension of the sketch on its Web page. More information about this can be seen in the reference for `size()`.

Processing variables that store the cursor position and the most recent key pressed are discussed in Input 1 (p. 205) and Input 2 (p. 223).

Exercises

1. *Think about different types of numbers you use daily. Are they integer or floating-point numbers?*
2. *Make a few `int` and `float` variables. Try assigning them in different ways. Write the values to the console with `println()`.*
3. *Create a composition that scales proportionally with different window sizes. Put different values into `size()` to test.*



Math 1: Arithmetic, Functions

This unit focuses on performing basic mathematical operations and using the results to control the position and properties of visual elements.

Syntax introduced:

```
+ (add), - (subtract), * (multiply), / (divide), % (modulus)
() (parentheses)
++ (increment), -- (decrement), += (add assign), -= (subtract assign)
*= (multiply assign), /= (divide assign), - (negation)
ceil(), floor(), round(), min(), max()
```

Math can be an important aspect of programming, but it's not necessary to be good at math to understand or enjoy programming. There are as many styles of programming as there are people who program, and it's the decision of the individual to utilize or ignore math as they prefer. People who enjoy math often write programs to visualize equations or take delight in exploring phenomena such as fractals. People who struggled with math in school sometimes find they enjoy and understand it better when it is applied to form and motion. This book discusses arithmetic, algebra, and trigonometry in the service of producing form and motion, a surprising range of which can be created with basic mathematics. A nuanced understanding of these techniques can yield more control over the software. More advanced mathematics such as linear algebra and calculus are often used for images and motion, but they fall outside the scope of this book.

Arithmetic

In programming, the visual properties of an image on the screen are defined by numbers, which means that the image can be controlled mathematically. For example, a rectangle might have a horizontal position of 10, a vertical position of 10, a width and height of 55, and a gray value of 153. If the gray value is stored in the variable *grayVal* and 102 is added to this variable, the gray value will become 255 and the shape will appear white on screen. This is demonstrated more succinctly in this example:

4-01

```
int grayVal = 153;
fill(grayVal);
rect(10, 10, 55, 55);     // Draw gray rectangle
grayVal = grayVal + 102;  // Assign 255 to grayVal
fill(grayVal);
rect(35, 30, 55, 55);     // Draw white triangle
```

The expression to the right of the = symbol is evaluated before the value is assigned to the variable on the left. Therefore, the statement `a = 5 + 4` first adds 5 and 4 to yield 9 and then assigns the value 9 to the variable `a`.

Within the visual domain, addition, subtraction, multiplication, and division can be used to control the position of elements on the screen or to change attributes such as size or gray value. The + symbol is used for addition, the - symbol for subtraction, the * symbol for multiplication, and the / symbol for division.

4-02

```
int a = 30;
line(a, 0, a, height);
a = a + 40;  // Assign 70 to a
strokeWeight(4);
line(a, 0, a, height);
```

4-03

```
int a = 30;
int b = 40;
line(a, 0, a, height);
line(b, 0, b, height);
strokeWeight(4);
// A calculation can be used as an input to a function
line(b-a, 0, b-a, height);
```

4-04

```
int a = 8;
int b = 10;
line(a, 0, a, height);
line(b, 0, b, height);
strokeWeight(4);
line(a*b, 0, a*b, height);
```

4-05

```
int a = 8;
int b = 10;
line(a, 0, a, height);
line(b, 0, b, height);
strokeWeight(4);
line(a/b, 0, a/b, height);
```

4-06

```
int y = 20;
line(0, y, width, y);
y = y + 6;  // Assign 26 to y
line(0, y, width, y);
y = y + 6;  // Assign 32 to y
line(0, y, width, y);
y = y + 6;  // Assign 38 to y
line(0, y, width, y);
```

```
float y = 20;
line(0, y, width, y);
y = y * 1.6;  // Assign 32.0 to y
line(0, y, width, y);
y = y * 1.6;  // Assign 51.2 to y
line(0, y, width, y);
y = y * 1.6;  // Assign 81.920006 to y
  line(0, y, width, y);
```

4-07

The +, -, *, /, and = symbols are probably familiar, but the % is more exotic. The % operator calculates the remainder when one number is divided by another. The %, the code notation for modulus, returns the integer remainder after dividing the number to the left of the symbol by the number to the right.

Expression	Result	Explanation
9 % 3	0	3 goes into 9 three times, with no remainder
9 % 2	1	2 goes into 9 four times, with 1 as the remainder
35 % 4	3	4 goes into 35 eight times, with 3 remaining

Modulus can be explained with an anecdote. After a hard day of work, Casey and Ben were hungry. They went to a restaurant to eat dumplings, but there were only 9 dumplings left so they had to share. If they share equally, how many dumplings can they each eat and how many will remain? It's obvious each can have 4 dumplings and 1 will remain. If there are 4 people and 35 dumplings, each can eat 8 and 3 will remain. In these examples, the modulus value is the number of remaining dumplings.

The modulus operator is often used to keep numbers within a desired range. For example, if a variable is continually increasing (0, 1, 2, 3, 4, 5, 6, 7, etc.), applying the modulus operator can transform this sequence. A continually increasing number can be made to cycle continuously between 0 and 3 by applying %4:

x	0	1	2	3	4	5	6	7	8	9	10
x % 4	0	1	2	3	0	1	2	3	0	1	2

Many examples throughout this book use % in this way.

When working with mathematical operators and variables, it's important to be aware of the data types of the variables you're using. The combination of two integers will always result in an `int`. The combination of two floating-point numbers will always result in a `float`, but when an `int` and `float` are operated on, the result is a `float`.

```
println(4/3);      // Prints "1"
println(4.0/3);    // Prints "1.3333334"
println(4/3.0);    // Prints "1.3333334"
println(4.0/3.0);  // Prints "1.3333334"
```

4-08

Integer values can be assigned to floating-point variables, but not vice versa. Assigning a `float` to an `int` makes the number less accurate, so Processing requires that you do so explicitly (discussed on page 105). Working with an `int` and a `float` will upgrade the `int` and treat both numbers as floating-point values, but the result won't fit into an `int` variable.

```
int a = 4/3;        // Assign 1 to a
int b = 3/4;        // Assign 0 to b
int c = 4.0/3;      // ERROR!
int d = 4.0/3.0;    // ERROR!
float e = 4.0/3;    // Assign 1.3333334 to e
float f = 4.0/3.0;  // Assign 1.3333334 to f
```

4-09

The last two calculations require additional explanation. The result of dividing two integers will always be an integer: dividing the integer 4 by the integer 3 equals 1. This result is converted to a floating-point variable *after* the calculation has finished, so the 1 becomes 1.0 only once it has reached the left side of the = sign. While this may seem confusing, it can be useful for more advanced programs.

```
float a = 4/3;      // Assign 1.0 to a
float b = 3/4;      // Assign 0.0 to b
```

4-10

The rules of calculating `int` and `float` values can become obscured when variables are used instead of the actual numbers. It's important to be aware of the data types for variables to avoid this problem.

```
int i = 4;
float f = 3.0;
int a = i/f;        // ERROR! Assign a float value to an int variable
float b = i/f;      // Assign 1.3333334 to b
```

4-11

It's also important to pay attention to the value of variables to avoid making arithmetic errors. For example, dividing a number by zero yields "infinity" in mathematics, but in software it just causes an error.

```
int a = 0;
int b = 12/a;       // ERROR! ArithmeticException: / by zero
```

4-12

Similarly, dividing by an extremely small number can yield an enormous result. This can be confusing when drawing shapes because they will not be visible in the display window.

```
float a = 0.0001;
float b = 12/a;  // Assign 120000.0 to b
```

4-13

Operator precedence, Grouping

The *order of operations* determines which math operators perform their calculations before others. For example, multiplication is always evaluated before addition regardless of the sequence of the elements. The expression `3 + 4 * 5` evaluates to 23 because 4 is first multiplied by 5 to yield 20 and then 3 is added to yield 23. The order of operations specifies that multiplication always precedes addition regardless of the order in which they appear in the expression. The order of operations for evaluating expressions may be changed by adding parentheses. For example, if an addition operation appears in parentheses, it will be performed prior to multiplication. The expression `(3 + 4) * 5` evaluates to 35 because 3 is first added to 4 to yield 7, which is then multiplied by 5 to yield 35. This is more concisely expressed in code:

```
x = 3 + 4 * 5;      // Assign 23 to x
y = (3 + 4) * 5;    // Assign 35 to y
```
4-14

In many cases, parentheses are necessary to force elements of an expression to evaluate before others, but sometimes they are used only to clarify the order of operations. The following lines calculate the same result because multiplication always happens before addition, but you may find the second line more clear.

```
x = 10 * 20 + 5;    // Assign 205 to x
y = (10 * 20) + 5;  // Assign 205 to y
```
4-15

The following table shows the operator precedence for the operators introduced so far. Items at the top precede those toward the bottom.

Multiplicative	`* / %`
Additive	`+ -`
Assignment	`=`

This means, for example, that division will always happen before subtraction and addition will always happen before assignment. A complete listing for the order of operations is listed in Appendix A (p. 661).

Shortcuts

There are many repetitive expressions in programming, so code shortcuts are used to make programs more concise. The increment operator ++ adds the value 1 to a variable and the decrement operator -- subtracts the value of 1:

```
int x = 1;                                                    4-16
println(x);  // Prints "1" to the console
x++;         // Equivalent to x = x + 1
println(x);  // Prints "2" to the console

int y = 1;
println(y);  // Prints "1" to the console
y--;         // Equivalent to y = y - 1
println(y);  // Prints "0" to the console
```

The value is incremented or decremented after the expression is evaluated. This often creates confusion and is shown in this example:

```
int x = 1;                                                    4-17
println(x++);  // Prints "1" to the console
println(x);    // Prints "2" to the console
```

To update the value before the expression is evaluated, place the operator in front of the variable:

```
int x = 1;                                                    4-18
println(++x);  // Prints "2" to the console
println(x);    // Prints "2" to the console
```

The add assign operator += combines addition and assignment. The subtract assign operator -= combines subtraction with assignment:

```
int x = 1;                                                    4-19
println(x);  // Prints "1" to the console
x += 5;      // Equivalent to x = x + 5
println(x);  // Prints "6" to the console

int y = 1;
println(y);  // Prints "1" to the console
y -= 5;      // Equivalent to y = y - 5
println(y);  // Prints "-4" to the console
```

The multiply assign operator *= combines multiplication with assignment. The divide assign operator /= combines division with assignment:

```
int x = 4;                                                    4-20
println(x);  // Prints "4" to the console
x *= 2;      // Equivalent to x = x * 2
println(x);  // Prints "8" to the console

int y = 4;
println(y);  // Prints "4" to the console
y /= 2;      // Equivalent to y = y / 2
println(y);  // Prints "2" to the console
```

The negation operator - changes the sign of value to its right. It can be used in place of multiplying a value by –1.

```
int x = 5;   // Assigns 5 to x                                 4-21
x = -x;      // Equivalent to x = x * -1
println(x);  // Prints "-5"
```

Constraining numbers

The `ceil()`, `floor()`, `round()`, `min()`, and `max()` functions are used to perform calculations on numbers that the standard arithmetic operators can't. These functions are different from those for drawing shapes, such as `line()` and `ellipse()`, because they return values. This means the function outputs a number that can be assigned to a variable.

The `ceil()` function calculates the closest `int` value that is greater than or equal to the value of its parameter:

```
int w = ceil(2.0);  // Assign 2 to w                           4-22
int x = ceil(2.1);  // Assign 3 to x
int y = ceil(2.5);  // Assign 3 to y
int z = ceil(2.9);  // Assign 3 to z
```

The `floor()` function calculates the closest `int` value that is less than or equal to the value of its parameter:

```
int w = floor(2.0);  // Assign 2 to w                          4-23
int x = floor(2.1);  // Assign 2 to x
int y = floor(2.5);  // Assign 2 to y
int z = floor(2.9);  // Assign 2 to z
```

The `round()` function calculates the `int` value closest to the value of its parameter. Values ending with .5 round up to the next `int` value:

```
int w = round(2.0);  // Assign 2 to w
int x = round(2.1);  // Assign 2 to x
int y = round(2.5);  // Assign 3 to y
int z = round(2.9);  // Assign 3 to z
```
4-24

Even though `ceil()`, `floor()`, and `round()` act on floating-point numbers, the result is always an integer, because that's the way the result will most often be useful. To convert to a `float`, simply assign the result to a float variable.

```
float w = round(2.1);  // Assign 2.0 to w
```
4-25

The `min()` function determines the smallest value in a sequence of numbers. The `max()` function determines the largest value in a sequence of numbers. Both functions can have two or three parameters:

```
int u = min(5, 9);            // Assign 5 to u
int v = min(-4, -12, -9);     // Assign -12 to v
float w = min(12.3, 230.24);  // Assign 12.3 to w
int x = max(5, 9);            // Assign 9 to x
int y = max(-4, -12, -9);     // Assign -4 to y
float z = max(12.3, 230.24);  // Assign 230.24 to z
```
4-26

Exercises

1. *Use one variable to set the position and size for three ellipses.*
2. *Use multiplication to create a series of lines with increasing space between each.*
3. *Explore the functions for constraining numbers. Use `min()` and `max()` to draw a regular pattern of lines from a sequence of irregular numbers.*

Control 1: Decisions

This unit focuses on controlling the flow of a program with conditional structures. Logical operators for extending relational expressions are introduced.

Syntax introduced:

```
> (greater than), < (less than)
>= (greater than or equal to), <= (less than or equal to)
== (equality), != (inequality)
if, else, {} (braces)
|| (logical OR), && (logical AND), ! (logical NOT)
```

The programs we've seen so far run each line of code in sequence. They run the first line, then the second, then the third, etc. The program stops when the last line is run. It's often beneficial to change this order—sometimes skipping lines or repeating lines many times to perform a repetitive action. Although the lines of code that comprise a program are always positioned in an order from top to bottom on the page, this doesn't necessarily define the order in which each line is run. This order is called the *flow* of the program. Flow can be changed by adding elements of code called control structures.

Relational expressions

What is truth? It's easy to answer this difficult philosophical question in the context of programming because the logical notions of true and false are well defined. Code elements called relational expressions evaluate to `true` and `false`. A relational expression is made up of two values that are compared with a relational operator. In Processing, two values can be compared with relational operators as follows:

Expression	Evaluation
`3 > 5`	`false`
`3 < 5`	`true`
`5 < 3`	`false`
`5 > 3`	`true`

Each of these statements can be converted to English. Using the first row as an example, we can say, "Is three greater than five?" The answer "no" is expressed with the value `false`. The next row can be converted to "Is three less than five?" The answer is "yes" and is expressed with the value `true`. A relational expression, two values compared with a relational operator, evaluates to `true` or `false`—there are no other possible values. The relational operators are defined as follows:

Operator	Meaning
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equivalent to
!=	not equivalent to

The following lines of code show the results of comparing the same group of numbers with different relational operators:

5-01

```
println(3 > 5);   // Prints "false"
println(5 > 3);   // Prints "true"
println(5 > 5);   // Prints "false"

println(3 < 5);   // Prints "true"
println(5 < 3);   // Prints "false"
println(5 < 5);   // Prints "false"

println(3 >= 5);  // Prints "false"
println(5 >= 3);  // Prints "true"
println(5 >= 5);  // Prints "true"

println(3 <= 5);  // Prints "true"
println(5 <= 3);  // Prints "false"
println(5 <= 5);  // Prints "true"
```

The equality operator, the == symbol, determines whether two values are equivalent. It is different from the = symbol, which assigns a value, but the two are often used erroneously in place of each other. The only way to avoid this mistake is to be careful. It's similar to using "their" instead of "there" when writing in English—a mistake that even experienced writers sometimes make. The != symbol is the opposite of == and determines whether two values are not equivalent.

5-02

```
println(3 == 5);  // Prints "false"
println(5 == 3);  // Prints "false"
println(5 == 5);  // Prints "true"

println(3 != 5);  // Prints "true"
println(5 != 3);  // Prints "true"
println(5 != 5);  // Prints "false"
```

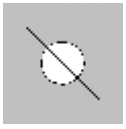
Conditionals

Conditionals allow a program to make decisions about which lines of code run and which do not. They let actions take place only when a specific condition is met. Conditionals allow a program to behave differently depending on the values of their variables. For example, the program may draw a line or an ellipse depending on the value of a variable. The `if` structure is used in Processing to make these decisions:

```
if (test) {
  statements
}
```

The test must be an expression that resolves to `true` or `false`. When the test expression evaluates to `true`, the code inside the { (left brace) and } (right brace) is run. If the expression is `false`, the code is ignored. Code inside a set of braces is called a block.

The following three examples present the same code with different values for the `x` variable. Because this variable is used in the test for the `if` structure, changing it affects which lines of code are run. Changing the value causes an ellipse, rectangle, or neither to draw to the display window.



5-03

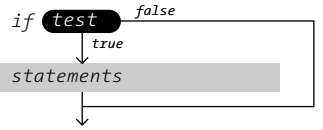
```
// The text expressions are "x > 100" and "x < 100"
// Because x is 150, the code inside the first block
// runs and the ellipse draws, but the code in the second
// block is not run and the rectangle is not drawn
int x = 150;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
}
if (x < 100) {              // If x is less than 100
  rect(35, 35, 30, 30);     // draw this rectangle
}
line(20, 20, 80, 80);
```

5-04

```
// Because x is 50, only the rectangle draws
int x = 50;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
}
if (x < 100) {              // If x is less than 100,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);
```

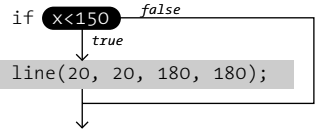
General case `if` structure

```
if (test) {
  statements
}
```



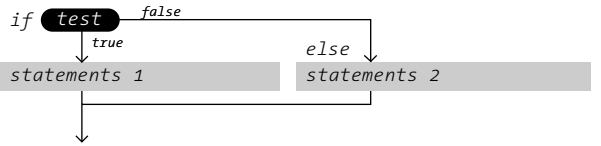
A specific `if` structure

```
if (x < 150) {
  line(20, 20, 180, 180);
}
```



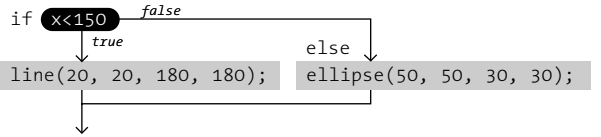
General case `if/else` structure

```
if (test) {
  statements 1
} else {
  statements 2
}
```



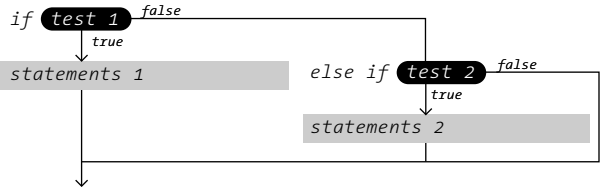
A specific `if/else` structure

```
if (x < 150) {
  line(20, 20, 180, 180);
} else {
  ellipse(50, 50, 30, 30);
}
```



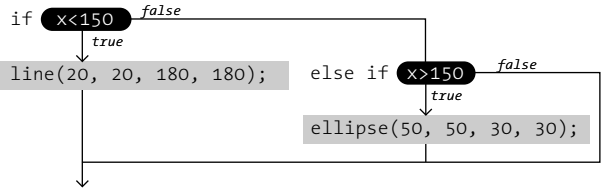
General case `if/else if` structure

```
if (test 1) {
  statements 1
} else if (test 2) {
  statements 2
}
```



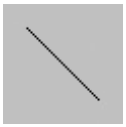
A specific `if/else if` structure

```
if (x < 150) {
  line(20, 20, 180, 180);
} else if (x > 150) {
  ellipse(50, 50, 30, 30);
}
```



Decisions

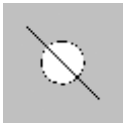
The flow of an `if`, `else`, and `else if` structure shown as a diagram. The code inside each block is run if the test evaluates to true. For each set of diagrams, the general case shows the generic format and the specific case shows one example of how the format can be used within a program.



```
// Because x is 100, only the line draws                    5-05
int x = 100;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
}
if (x < 100) {              // If x is less than 100,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);       // Always draw the line
```

To run a different set of code when the relational expression for an `if` structure is not `true`, use the `else` keyword. The keyword `else` extends an `if` structure so that when the expression associated with the structure is `false`, the code in the `else` block is run instead.

```
//  Because x is 90, only the rectangle draws                5-06
int x = 90;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
} else {                    // Otherwise,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);       // Always draw the line
```



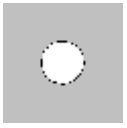
```
// Because x is 290, only the ellipse draws                 5-07
int x = 290;
if (x > 100) {              // If x is greater than 100,
  ellipse(50, 50, 36, 36);  // draw this ellipse
} else {                    // Otherwise,
  rect(33, 33, 34, 34);     // draw this rectangle
}
line(20, 20, 80, 80);       // Always draw the line
```

Conditionals can be embedded within other conditionals to control which lines of code will run. In the next example, the code for drawing the ellipse or line can be reached only if `x` is larger than 100. If this expression evaluates to `true`, a second comparison of `x` determines which of these shapes will be drawn.

5-08

```
// If x is greater than 100 and less than 300, draw the
// ellipse. If x is greater than or equal to 300, draw
// the line. If x is not greater than 100, draw the
// rectangle. Because x is 420, only the line draws.
int x = 420;
if (x > 100) {    // First test to draw ellipse or line
  if (x < 300) {  // Second test determines which to draw
    ellipse(50, 50, 36, 36);
  } else {
    line(50, 0, 50, 100);
  }
} else {
  rect(33, 33, 34, 34);
}
```

Conditionals can be extended further by combining an `else` with an `if`. This allows conditionals to use multiple tests to determine which lines the program should run. This technique is used when there are many choices and only one can be selected at a time.



5-09

```
// If x is less than or equal to 100, then draw
// the rectangle. Otherwise, if x is greater than
// or equal to 300, draw the line. If x is between
// 100 and 300, draw the ellipse. Because x is 101,
// only the ellipse draws.
int x = 101;
if (x <= 100) {
  rect(33, 33, 34, 34);
} else if (x >= 300) {
  line(50, 0, 50, 100);
} else {
  ellipse(50, 50, 36, 36);
}
```

Logical operators

Logical operators are used to combine two or more relational expressions and to invert logical values. They allow for more than one condition to be considered simultaneously. The logical operators are symbols for the logical concepts of AND, OR, and NOT:

Operator	Meaning
&&	AND
||	OR
!	NOT

The following table outlines all possible combinations and the results.

Expression	Evaluation
`true && true`	`true`
`true && false`	`false`
`false && false`	`false`
`true || true`	`true`
`true || false`	`true`
`false || false`	`false`
`!true`	`false`
`!false`	`true`

The logical OR operator, two vertical bars (sometimes called pipes), makes the relational expression `true` if only one part is `true`. The following example shows how to use it:

5-10

```
int a = 10;
int b = 20;
// The expression "a > 5" must be true OR "b < 30"
// must be true. Because they are both true, the code
// in the block will run.
if ((a > 5) || (b < 30)) {
  line(20, 50, 80, 50);
}
// The expression "a > 15" is false, but "b < 30"
// is true. Because the OR operator requires only one part
// to be true in the entire expression, the code in the
// block will run.
if ((a > 15) || (b < 30)) {
  ellipse(50, 50, 36, 36);
}
```

Compound logical expressions can be tricky to figure out, but they are simpler when looked at step by step. Parentheses are useful hints in determining the order of

evaluation. Looking at the test of the `if` structure in line 6 of the previous example, first the variables are replaced with their values, then each subexpression is evaluated, and finally the expression with the logical operator is evaluated:

```
Step 1      (a > 5) || (b < 30)
Step 2      (10 > 5) || (20 < 30)
Step 3      true || true
Step 4      true
```

The logical AND operator, two ampersands, allows the entire relational statement to be `true` only if both parts are `true`. The following example is the same as the last except the logical OR operators have been changed to the logical AND. Because each operator compares the values differently, only the line is drawn here, whereas the previous example drew both the line and circle.

5-11

```
int a = 10;
int b = 20;
// The expression "a > 5" must be true AND "b < 30"
// must be true. Because they are both true, the code
// in the block will run.
if ((a > 5) && (b < 30)) {
  line(20, 50, 80, 50);
}
// The expression "a > 15" is false, but "b < 30" is
// true. Because the AND operator requires both to be
// true, the code in the block will not run.
if ((a > 15) && (b < 30)) {
  ellipse(50, 50, 36, 36);
}
```

Technically, the steps shown above aren't the whole story. When using AND, the first part of the expression will be evaluated. If that part is `false`, then the second part of the expression won't even be evaluated. For example, in this expression . . .

```
(a > 5) && (b < 30)
```

. . . if `a > 5` evaluates to `false`, then `b < 30` is ignored for efficiency. This is called a *short circuit* operator. The same happens for the OR operator, where the first `true` statement will end evaluation. For example, if the expression is:

```
(a > 5) || (b < 30)
```

If `a > 5` is true, then the `b < 30` will be ignored, because the entire expression will evaluate to true, regardless of the value of `b < 30`. Outside of efficiency, this has many

practical applications in more advanced code.

The logical NOT operator is an exclamation mark. It inverts the logical value of the associated `boolean` variables. It changes `true` to `false`, and `false` to `true`. The logical NOT operator can be applied only to `boolean` variables.

5-12

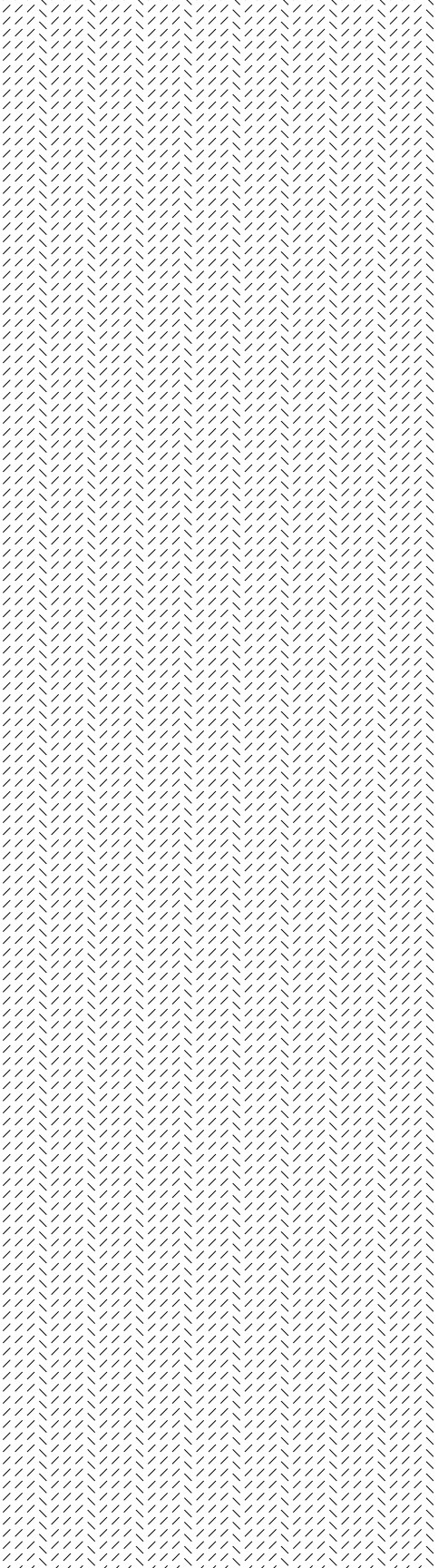
```
boolean b = true;   // Assign true to b
println(b);         // Prints "true"
println(!b);        // Prints "false"
b = !b;             // Assign false to b
println(b);         // Prints "false"
println(!b);        // Prints "true"
println(5 > 3);     // Prints "true"
println(!(5 > 3));  // Prints "false"
int x = 5;
println(!x);  // ERROR! It's only possible to ! a boolean variable
```

5-13

```
// Because b is true, the line draws
boolean b = true;
if (b == true) {            // If b is true,
  line(20, 50, 80, 50);     // draw the line
}
if (!b == true) {           // If b is false,
  ellipse(50, 50, 36, 36);  // draw the ellipse
}
```

Exercises

1. *Create a few relational expressions and print their evaluation to the console with* `println()`.
2. *Create a composition with a series of lines and ellipses. Use an* `if` *structure to select which lines of code to run and which to skip.*
3. *Add an* `else` *to the code from exercise 2 to change which code is run.*



Control 2: Repetition

This unit focuses on controlling the flow of programs with iterative structures.

Syntax introduced:
`for`

The early history of computers is the history of automating calculation. A "computer" was originally a person who was paid to calculate math by hand. What we know as a computer today emerged from machines built to automate tedious mathematical calculations. The earliest mechanical computers were calculators developed for speed and accuracy in performing repetitive calculations. Because of this heritage, computers are excellent at executing repetitive tasks accurately and quickly. Modern computers are also logic machines. Building on the work of the logicians Leibniz and Boole, modern computers use logical operations such as AND, OR, and NOT to determine which lines of code are run and which are not.

Iteration

Iterative structures are used to compact lengthy lines of repetitive code. Decreasing the length of the code can make programs easier to manage and can also help to reduce errors. The table below shows equivalent programs written without an iterative structure and with a `for` structure. The original 14 lines of code on the left are reduced to the 4 lines on the right:

Original code

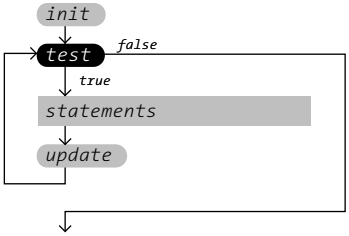
```
size(200, 200);
line(20, 20, 20, 180);
line(30, 20, 30, 180);
line(40, 20, 40, 180);
line(50, 20, 50, 180);
line(60, 20, 60, 180);
line(70, 20, 70, 180);
line(80, 20, 80, 180);
line(90, 20, 90, 180);
line(100, 20, 100, 180);
line(110, 20, 110, 180);
line(120, 20, 120, 180);
line(130, 20, 130, 180);
line(140, 20, 140, 180);
```

Code expressed using a `for` structure

```
size(200, 200);
for (int i = 20; i < 150; i += 10) {
  line(i, 20, i, 180);
}
```

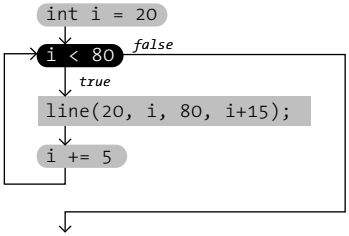
General case `for` structure

```
for (init; test; update) {
  statements
}
```



A specific `for` structure

```
for (int i = 20; i < 80; i += 5) {
  line(20, i, 80, i+15);
}
```



Repetition

The flow of a `for` structure shown as a diagram. These images show the central importance of the test statement in deciding whether to run the code in the block or to exit. The general case shows the generic format, and the specific case shows one example of how the format can be used within a program.

The `for` structure performs repetitive calculations and is structured like this:

```
for (init; test; update) {
  statements
}
```

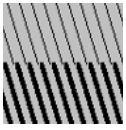
The parentheses associated with the structure enclose three statements: *init*, *test*, and *update*. The statements inside the block are run continuously while the test evaluates to `true`. The *init* portion assigns the initial value of the variable used in the test. The *update* is used to modify the variable after each iteration through the block. A `for` structure runs in the following sequence:

1. The *init* statement is run
2. The *test* is evaluated to `true` or `false`
3. If the *test* is `true`, continue to step 4. If the *test* is `false`, jump to step 6
4. Run the statements within the block
5. Run the *update* statement and jump to step 2
6. Exit the structure and continue running the program

The following examples demonstrate how the `for` structure is used within a program to control the way shapes are drawn to the display window.

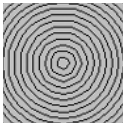
6-01

```
// The init is "int i = 20", the test is "i < 80",
// and the update is "i += 5". Notice the semicolon
// terminating the first two elements
for (int i = 20; i < 80; i += 5) {
  // This line will continue to run until "i"
  // is greater than or equal to 80
  line(20, i, 80, i+15);
}
```



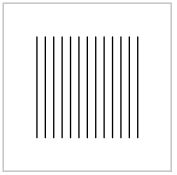
6-02

```
for (int x = -16; x < 100; x += 10) {
  line(x, 0, x+15, 50);
}
strokeWeight(4);
for (int x = -8; x < 100; x += 10) {
  line(x, 50, x+15, 100);
}
```

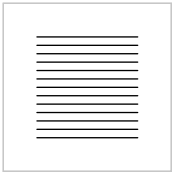


6-03

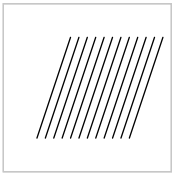
```
noFill();
for (int d = 150; d > 0; d -= 10) {
  ellipse(50, 50, d, d);
}
```



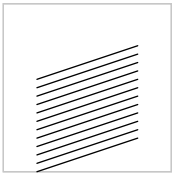
```
for (int x = 20; x <= 80; x += 5) {
  line(x, 20, x, 80);
}
```



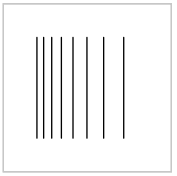
```
for (int x = 20; x <= 80; x += 5) {
  line(20, x, 80, x);
}
```



```
for (int x = 20; x < 80; x += 5) {
  line(x+20, 20, x, 80);
}
```

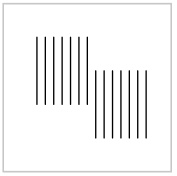


```
for (float x = 80; x > 20; x -= 5) {
  line(20, x+20, 80, x);
}
```

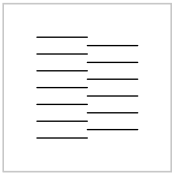


```
for (float x = 20; x < 80; x *= 1.2) {
  line(x, 20, x, 80);
}
```

```
for (float x = 80; x > 20; x /= 1.2) {
  line(20, x, 80, x);
}
```



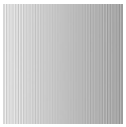
```
for (int x = 20; x <= 85; x += 5) {
  if (x <= 50) {
    line(x, 20, x, 60);
  } else {
    line(x, 40, x, 80);
  }
}
```



```
for (int x = 20; x <= 80; x += 5) {
  if ((x % 10) == 0) {
    line(20, x, 50, x);
  } else {
    line(50, x, 80, x);
  }
}
```

All for one and one for all

The* `for` *structure is flexible, but it always follows the rules. These examples show how it can be used to generate various patterns.



```
for (int i = 0; i < 100; i += 2) {
  stroke(255-i);
  line(i, 0, i, 200);
}
```

6-04

Nested iteration

The `for` structure produces repetitions in one dimension. Nesting one of these structures into another compounds their effect, creating iteration in two dimensions. Instead of drawing 9 points and then drawing another 9 points, they combine to create 81 points; for each point drawn in the outer structure, 9 points are drawn in the inner structure. The inner structure runs through a complete cycle for each single iteration of the outer structure. In the following examples, the two dimensions are translated into x-coordinates and y-coordinates:

```
for (int y = 10; y < 100; y += 10) {
  point(10, y);
}
```

6-05

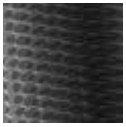
```
for (int x = 10; x < 100; x += 10) {
  point(x, 10);
}
```

6-06

```
for (int y = 10; y < 100; y += 10) {
  for (int x = 10; x < 100; x += 10) {
    point(x, y);
  }
}
```

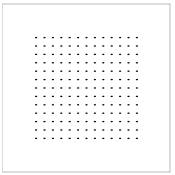
6-07

This technique is useful for creating diverse patterns and effects. The numbers produced by embedding iterative elements can be applied to color, position, size, transparency, and any other visual attribute.



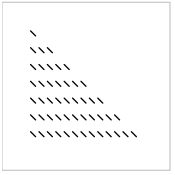
```
fill(0, 76);
noStroke();
smooth();
for (int y = -10; y <= 100; y += 10) {
  for (int x = -10; x <= 100; x += 10) {
    ellipse(x + y/8.0, y + x/8.0, 15 + x/2, 10);
  }
}
```

6-08

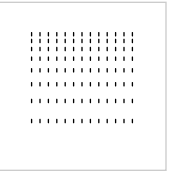


```
for (int y = 20; y <= 80; y += 5) {
  for (int x = 20; x <= 80; x += 5) {
    point(x, y);
  }
}
```

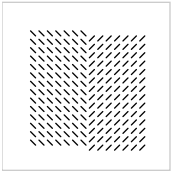
```
for (int y = 20; y <= 80; y += 3) {
  for (int x = 20; x <= 80; x += 10) {
    point(x, y);
  }
}
```



```
for (int y = 20; y <= 80; y += 10) {
  for (int x = 20; x <= y; x += 5) {
    line(x, y, x-3, y-3);
  }
}
```



```
for (float y = 20; y <= 80; y *= 1.2) {
  for (int x = 20; x <= 80; x += 5) {
    line(x, y, x, y-2);
  }
}
```

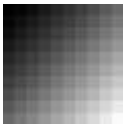


```
for (int y = 20; y <= 85; y += 5) {
  for (int x = 20; x <= 85; x += 5) {
    if (x <= 50) {
      line(x, y, x-3, y-3);
    } else {
      line(x, y, x-3, y+3);
    }
  }
}
```

```
for (int y = 20; y <= 80; y += 5) {
  for (int x = 20; x <= 80; x += 5) {
    if ((x % 10) == 0) {
      line(x, y, x+3, y-3);
    } else {
      line(x, y, x+3, y+3);
    }
  }
}
```

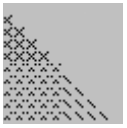
Embedding (nesting)

Embedding one `for` structure inside another is a highly malleable technique for drawing patterns. These examples show only a few of the possible options.



```
noStroke();
for (int y = 0; y < 100; y += 10) {
  for (int x = 0; x < 100; x += 10) {
    fill((x+y) * 1.4);
    rect(x, y, 10, 10);
  }
}
```

6-09



```
for (int y = 1; y < 100; y += 10) {
  for (int x = 1; x < y; x += 10) {
    line(x, y, x+6, y+6);
    line(x+6, y, x, y+6);
  }
}
```

6-10

Formatting code blocks

It's important to space code so the blocks are clear. The lines inside a block are typically offset to the right with spaces or tabs. When programs become longer, clearly defining the beginning and end of the block reveals the structure of the program and makes it more legible. This is the convention used in this book:

```
int x = 50;

if (x > 100) {
  line(20, 20, 80, 80);
} else {
  line(80, 20, 20, 80);
}
```

6-11

This is an alternative format that is sometimes used elsewhere:

```
int x = 50;

if (x > 100)
{
  line(20, 20, 80, 80);
}
else
{
  line(20, 80, 80, 20);
}
```

6-12

It's essential to use formatting to show the hierarchy of your code. The Processing environment will attempt basic formatting as you type, and you can use the "Auto Format" function from the Tools menu to clean up your code at any time. The `line()` function in the following code fragment is inside the `if` structure, but the spacing does not reveal this at a quick glance. Avoid formatting code like this:

```
int x = 50;                                                    6-13

if (x > 100) {
line(20, 20, 80, 80);  // Avoid formatting code like this
} else {               // because it makes it difficult to see
line(80, 20, 20, 80);  // what is inside the block
}
```

Exercises

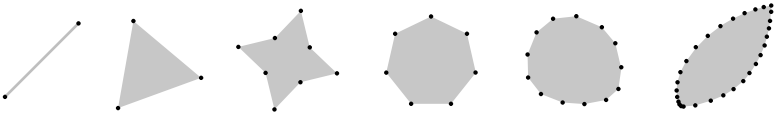
1. *Draw a regular pattern with five lines. Rewrite the code using a `for` structure.*
2. *Draw a dense pattern by embedding two `for` structures.*
3. *Combine two relational expressions with a logical operator to control the form of a pattern.*

Shape 2: Vertices

This unit focuses on drawing lines and shapes from sequences of vertices.

Syntax introduced:
`beginShape(), endShape(), vertex()`
`curveVertex(), bezierVertex()`

The geometric primitives introduced in Shape 1 provide extraordinary visual potential, but a programmer may often desire more complex shapes. Fortunately, there are many ways to define visual form with software. This unit introduces a way to define shapes as a series of coordinates, called vertices. A vertex is a position defined by an x- and y-coordinate. A line has two vertices, a triangle has three, a quadrilateral has four, and so on. Organic shapes such as blobs or the outline of a leaf are constructed by positioning many vertices in spatial patterns:



These shapes are simple compared to the possibilities. In contemporary video games, for example, highly realistic characters and environmental elements may be made up of more than 15,000 vertices. They represent more advanced uses of this technique, but they are created using similar principles.

Vertex

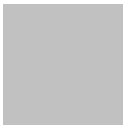
To create a shape from vertex points, first use the `beginShape()` function, then specify a series of points with the `vertex()` function and complete the shape with `endShape()`. The `beginShape()` and `endShape()` functions must always be used in pairs. The `vertex()` function has two parameters to define the x-coordinate and y-coordinate:

```
vertex(x, y)
```

By default, all shapes drawn with the `vertex()` function are filled white and have a black outline connecting all points except the first and last. The `fill()`, `stroke()`, `noFill()`, `noStroke()`, and `strokeWeight()` functions control the attributes of shapes drawn with the `vertex()` function, just as they do for those drawn with the shape functions discussed in Shape 1 (p. 23). To close the shape, use the `CLOSE` constant as a parameter for `endShape()`.

```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

7-01



```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape(CLOSE);
```

7-02

The order of the vertex positions changes the way the shape is drawn. The following example uses the same vertex positions as code 7-01, but the order of the third and fourth points are reversed.

```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(30, 75);
vertex(85, 75);
endShape();
```

7-03

Adding more vertex points reveals more of the potential of these functions. The following examples show variations of turning off the fill and stroke attributes and embedding `vertex()` functions within a `for` structure.

```
fill(0);
noStroke();
smooth();
beginShape();
vertex(10, 0);
vertex(100, 30);
vertex(90, 70);
vertex(100, 70);
vertex(10, 90);
vertex(50, 40);
endShape();
```

7-04

7-05

```
noFill();
smooth();
strokeWeight(20);
beginShape();
vertex(52, 29);
vertex(74, 35);
vertex(60, 52);
vertex(61, 75);
vertex(40, 69);
vertex(19, 75);
endShape();
```

7-06

```
noStroke();
fill(0);
beginShape();
vertex(40, 10);
for (int i = 20; i <= 100; i += 5) {
  vertex(20, i);
  vertex(30, i);
}
vertex(40, 100);
endShape();
```

A shape can have thousands of vertex points, but drawing too many points can slow down your programs.

Points, Lines

The `beginShape()` function can accept different parameters to define what to draw from the vertex data. The same points can be used to create a series of points, an unfilled shape, or a continuous line. The parameters `POINTS` and `LINES` are used to create different configurations of points and lines from the coordinates defined in the `vertex()` functions. Remember to type these parameters in uppercase letters because Processing is case-sensitive (p. 20).

7-07

```
// Draws a point at each vertex
beginShape(POINTS);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```
// Draws a line between each pair of vertices
beginShape(LINES);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

7-08

Shapes

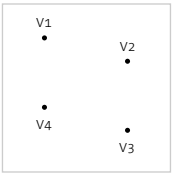
Use the parameters TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, and QUAD_STRIP with beginShape() to create other kinds of shapes. It's important to be aware of the spatial order of the vertex points when using these parameters because they affect how a shape is rendered. If the order required for each parameter is not followed, the expected shape will not draw. It's easy to change between working with TRIANGLES and a TRIANGLE_STRIP because the vertices can remain in the same spatial order, but this is not the case for changing between QUADS and a QUAD_STRIP. Refer to the examples below and the facing diagram for more information.

```
// Connects each grouping of three vertices as a triangle
beginShape(TRIANGLES);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```

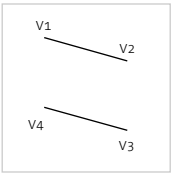
7-09

```
// Starting with the third vertex, connects each
// subsequent vertex to the previous two
beginShape(TRIANGLE_STRIP);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```

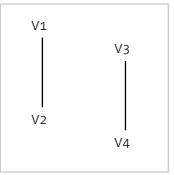
7-10



POINTS

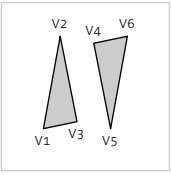


LINES

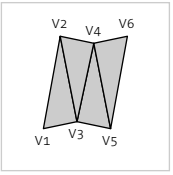


LINES

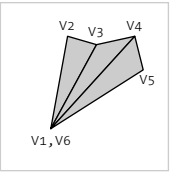
POINTS, LINES
The same data can be interpreted as a sequence of points or lines. The spatial order of the points affects what is drawn when using `LINES`.



TRIANGLES

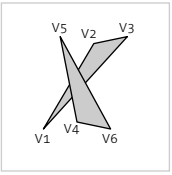


TRIANGLE_STRIP

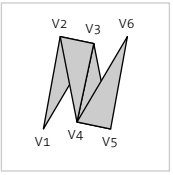


TRIANGLE_FAN

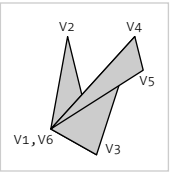
TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP
Groups of three vertices are drawn as individual triangles or a connected group.



TRIANGLES

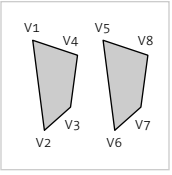


TRIANGLE_STRIP

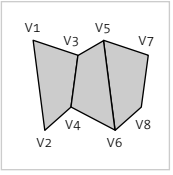


TRIANGLE_FAN

Unexpected results occur if the defined order is not followed.

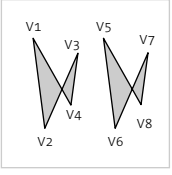


QUADS

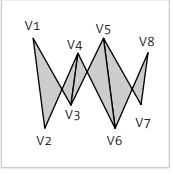


QUAD_STRIP

QUADS, QUAD_STRIP
Groups of four vertices are drawn as individual quads or a connected group. The spatial order determines whether a quad or a "bow" is drawn. Note that the order is reversed for `QUADS` *and* `QUAD_STRIP`.



QUADS



QUAD_STRIP

Parameters for `beginShape()`
There are eight options for the `MODE` *parameter of the* `beginShape()` *function, and each interprets vertex data in a different way. The notation V1, V2, V3, etc., represents the order and position of each vertex point.*

```
beginShape(TRIANGLE_FAN);
vertex(10, 20);
vertex(75, 30);
vertex(75, 50);
vertex(90, 70);
vertex(10, 20);
endShape();
```

7-11

```
beginShape(QUADS);
vertex(30, 25);
vertex(85, 30);
vertex(85, 50);
vertex(30, 45);
vertex(30, 60);
vertex(85, 65);
vertex(85, 85);
vertex(30, 80);
endShape();
```

7-12

```
// Notice the different vertex order for
// this example in relation to example 7-12
beginShape(QUAD_STRIP);
vertex(30, 25);
vertex(85, 30);
vertex(30, 45);
vertex(85, 50);
vertex(30, 60);
vertex(85, 65);
vertex(30, 80);
vertex(85, 85);
endShape();
```

7-13

Curves

The `vertex()` function works well for drawing straight lines, but if you want to create shapes made of curves, the two functions `curveVertex()` and `bezierVertex()` can be used to connect points with curves. These functions can be run between `beginShape()` and `endShape()` only when `beginShape()` has no parameter.

The `curveVertex()` function is used to set a series of points that connect with a curve. It has two parameters that set the x-coordinate and y-coordinate of the vertex.

```
curveVertex(x, y)
```

The first and last `curveVertex()` within a `beginShape()` and `endShape()` act as

control points, setting the curvature for the beginning and end of the line. The curvature for each segment of the curve is calculated from each pair of points in consideration of points before and after. Therefore, there must be at least four `curveVertex()` functions within `beginShape()` and `endShape()` to draw a segment.

7-14

```
smooth();
noFill();
beginShape();
curveVertex(20, 80);  // C1 (see p.76)
curveVertex(20, 40);  // V1
curveVertex(30, 30);  // V2
curveVertex(40, 80);  // V3
curveVertex(80, 80);  // C2
endShape();
```

Each `bezierVertex()` defines the position of two control points and one anchor point of a Bézier curve:

```
bezierVertex(cx1, cy1, cx2, cy2, x, y)
```

The first time `bezierVertex()` is used within `beginShape()`, it must be prefaced with `vertex()` to set the first anchor point. The line is drawn between the point defined by `vertex()` and the point defined by the x and y parameters to `bezierVertex()`. The first four parameters to the function position the control points to define the shape of the curve. The curve from code 2-21 (p. 30) was converted to this technique to yield the following example:

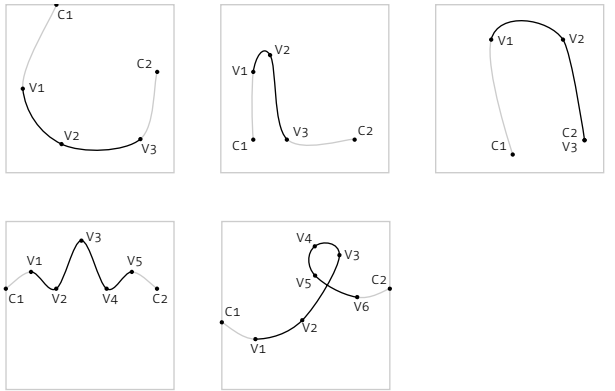
7-15

```
noFill();
beginShape();
vertex(32, 20);                          // V1 (see p.76)
bezierVertex(80, 5,  80, 75,  30, 75);   // C1, C2, V2
endShape();
```

Long, continuous curves can be made with `bezierVertex()`. After the first `vertex()` and `bezierVertex()`, each subsequent call to the function continues the shape by connecting each new point to the previous point.

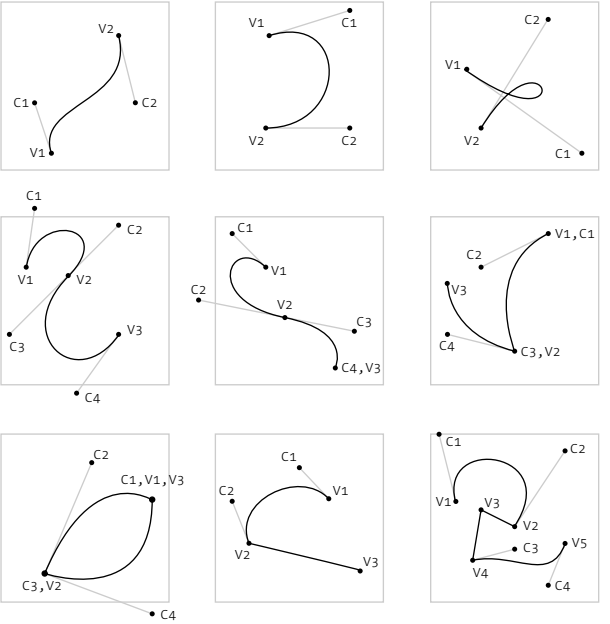
7-16

```
smooth();
noFill();
beginShape();
vertex(15, 30);                          // V1 (see p.76)
bezierVertex(20, -5, 70, 5, 40, 35);     // C1, C2, V2
bezierVertex(5, 70, 45, 105, 70, 70);    // C3, C4, V3
endShape();
```



Curve vertices

The `curveVertex()` function defines coordinates that are connected with curved shapes. The first and last points are control points that define the shape of the curve at the end and beginning.



Bézier vertices

Bézier curves are defined by vertex points and control points used as parameters to the `bezierVertex()` function. The control points define the shape of the curves that are drawn between the vertex points.

Curves

These curves are converted to software with the `vertex()`, `curveVertex()`, and `bezierVertex()` functions. The notation V0, V1, V2, etc., represents the order and position of each vertex point, and the notation C1, C2, C3, etc., represents the control points. Some of these curves are translated to software in codes 7-14 to 7-18.

To make a sharp turn, use the same position to specify the vertex and the following control point. To close the shape, use the same position to specify the first and last vertex.

7-17

```
smooth();
noStroke();
beginShape();
vertex(90, 39);                          // V1 (see p.76)
bezierVertex(90, 39, 54, 17, 26, 83);    // C1, C2, V2
bezierVertex(26, 83, 90, 107, 90, 39);   // C3, C4, V3
endShape();
```

Place the `vertex()` function within `bezierVertex()` functions to break the sequence of curves and draw a straight line.

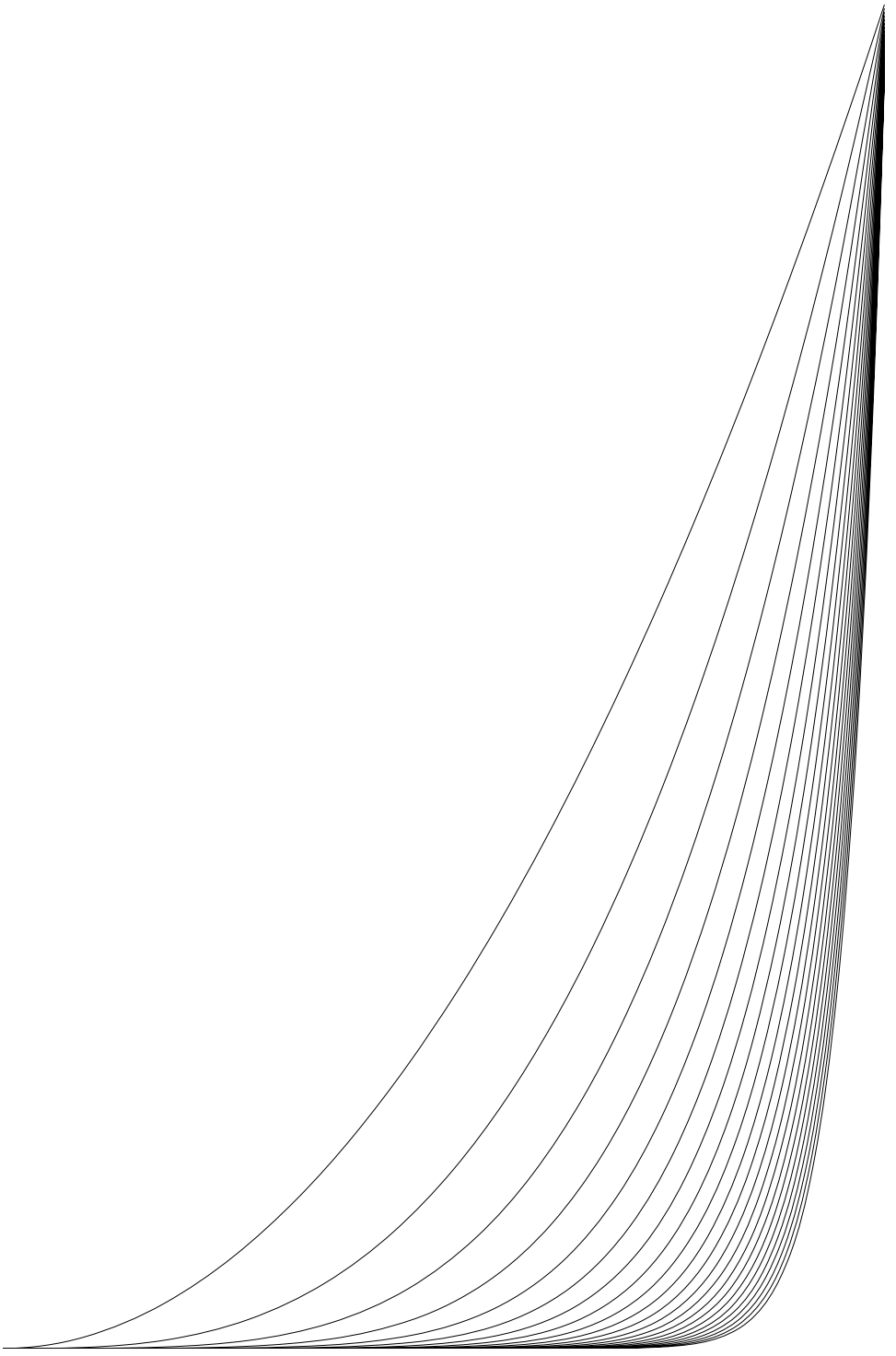
7-18

```
smooth();
noFill();
beginShape();
vertex(15, 40);                          // V1 (see p.76)
bezierVertex(5, 0, 80, 0, 50, 55);       // C1, C2, V2
vertex(30, 45);                          // V3
vertex(25, 75);                          // V4
bezierVertex(50, 70, 75, 90, 80, 70);    // C3, C4, V5
endShape();
```

A good technique for creating complex shapes with `beginShape()` and `endShape()` is to draw them first in a vector drawing program such as Inkscape or Illustrator. The coordinates can be read as numbers in this environment and then used in Processing. Another strategy for drawing intricate shapes is to create them in a vector-drawing program and then import the coordinates as a file. Processing includes a simple library for reading SVG files. Other libraries that support more formats and greater complexity can be found on the Processing website at *www.processing.org/reference/libraries*.

Exercises

1. *Use `beginShape()` to draw a shape of your own design.*
2. *Use different parameters for `beginShape()` to change the way a series of vertices are drawn.*
3. *Draw a complex curved shape of your own design using `bezierVertex()`.*



Math 2: Curves

This unit introduces drawing curves with mathematical equations.

Syntax introduced:
`sq(), sqrt(), pow(), norm(), lerp(), map()`

Basic mathematical equations can be used to draw shapes to the screen and modify their attributes. These equations augment the drawing functions discussed in Shape 1 (p. 23) and Shape 2 (p. 69). They can control movement and the way elements respond to the cursor. This math is used to accelerate and decelerate shapes in motion and move objects along curved paths.

Exponents, Roots

The `sq()` function is used to square a number and return the result. The result is always a positive number, because multiplying two negative numbers yields a positive result. For example, `-1 * -1 = 1`. This function has one parameter:

```
sq(value)
```

The value parameter can be any number. When `sq()` is used, the result can be assigned to a variable:

```
float a = sq(1);   // Assign 1 to a: Equivalent to 1 * 1
float b = sq(-5);  // Assign 25 to b: Equivalent to -5 * -5
float c = sq(9);   // Assign 81 to c: Equivalent to 9 * 9
```

8-01

The `sqrt()` function is used to calculate the square root of a number and return the result. It is the opposite of `sq()`. The square root of a number is always positive, even though there may be a valid negative root. The square root `s` of number `a` satisfies the equation `s * s = a`. This function has one parameter which must be a positive number:

```
sqrt(value)
```

As in the `sq()` function, the *value* parameter can be any number, and when the function is used the result can be assigned to a variable:

```
float a = sqrt(6561);  // Assign 81 to a
float b = sqrt(625);   // Assign 25 to b
float c = sqrt(1);     // Assign 1 to c
```

8-02

The pow() function calculates a number raised to an exponent. It has two parameters:

```
pow(num, exponent)
```

The *num* parameter is the number to multiply, and the *exponent* parameter is the number of times to make the calculation. The following example shows how it is used:

8-03

```
float a = pow(1, 3);   // Assign 1.0 to a: Equivalent to 1*1*1
float b = pow(3, 4);   // Assign 81.0 to b: Equivalent to 3*3*3*3
float c = pow(3, -2);  // Assign 0.11 to c: Equivalent to 1 / 3*3
float d = pow(-3, 3);  // Assign -27.0 to d: Equivalent to -3*-3*-3
```

Any number (except 0) raised to the zero power equals 1. Any number raised to the power of one equals itself.

8-04

```
float a = pow(8, 0);  // Assign 1 to a
float b = pow(3, 1);  // Assign 3 to b
float c = pow(4, 1);  // Assign 4 to c
```

Normalizing, Mapping

Numbers are often converted to the range 0.0 to 1.0 for making calculations. This is called *normalizing* the values. When numbers between 0.0 and 1.0 are multiplied together, the result is never less than 0.0 or greater than 1.0. This allows any number to be multiplied by another or by itself many times without leaving this range. For example, multiplying the value 0.2 by itself 5 times (0.2 * 0.2 * 0.2 * 0.2 * 0.2) produces the result 0.00032. Because normalized numbers have a decimal point, all calculations should be made with the float data type.

To normalize a number, divide it by the maximum value that it represents. For example, to normalize a series of values between 0.0 and 255.0, divide each by 255.0:

Initial value	Calculation	Normalized value
0.0	0.0 / 255.0	0.0
102.0	102.0 / 255.0	0.4
255.0	255.0 / 255.0	1.0

This can also be accomplished via the norm() function. It has three parameters:

```
norm(value, low, high)
```

The number used as the *value* parameter is converted to a value between 0.0 and 1.0. The *low* and *high* parameters set the respective minimum and maximum values of the

number's current range. If *value* is outside the range, the result may be less than 0 or greater than 1. The following example shows how to use the function to make the same calculations as the above table.

8-05

```
float x = norm(0.0, 0.0, 255.0);    // Assign 0.0 to x
float y = norm(102.0, 0.0, 255.0);  // Assign 0.4 to y
float z = norm(255.0, 0.0, 255.0);  // Assign 1.0 to z
```

After normalization, a number can be converted to another range through arithmetic. For example, to convert numbers between 0.0 and 1.0 in a range between 0.0 and 500.0, multiply by 500.0. To put numbers between 0.0 and 1.0 to numbers between 200.0 and 250.0, multiply by 50 then add 200. The following table presents a few sample conversions. The parentheses are used to improve readability:

Initial range of x	Desired range of x	Conversion
0.0 to 1.0	0.0 to 255.0	x * 255.0
0.0 to 1.0	-1.0 to 1.0	(x * 2.0) - 1.0
0.0 to 1.0	-20.0 to 60.0	(x * 80.0) - 20.0

The `lerp()` function can be used to accomplish these calculations. The name "lerp" is a contraction for "linear interpolation." The function has three parameters:

```
lerp(value1, value2, amt)
```

The *value1* and *value2* parameters define the minimum and maximum values and the *amt* parameter defines the value to interpolate between the values. The *amt* parameter should always be a value between 0.0 and 1.0. The following example shows how to use `lerp()` to make the value conversions on the last line of the previous table.

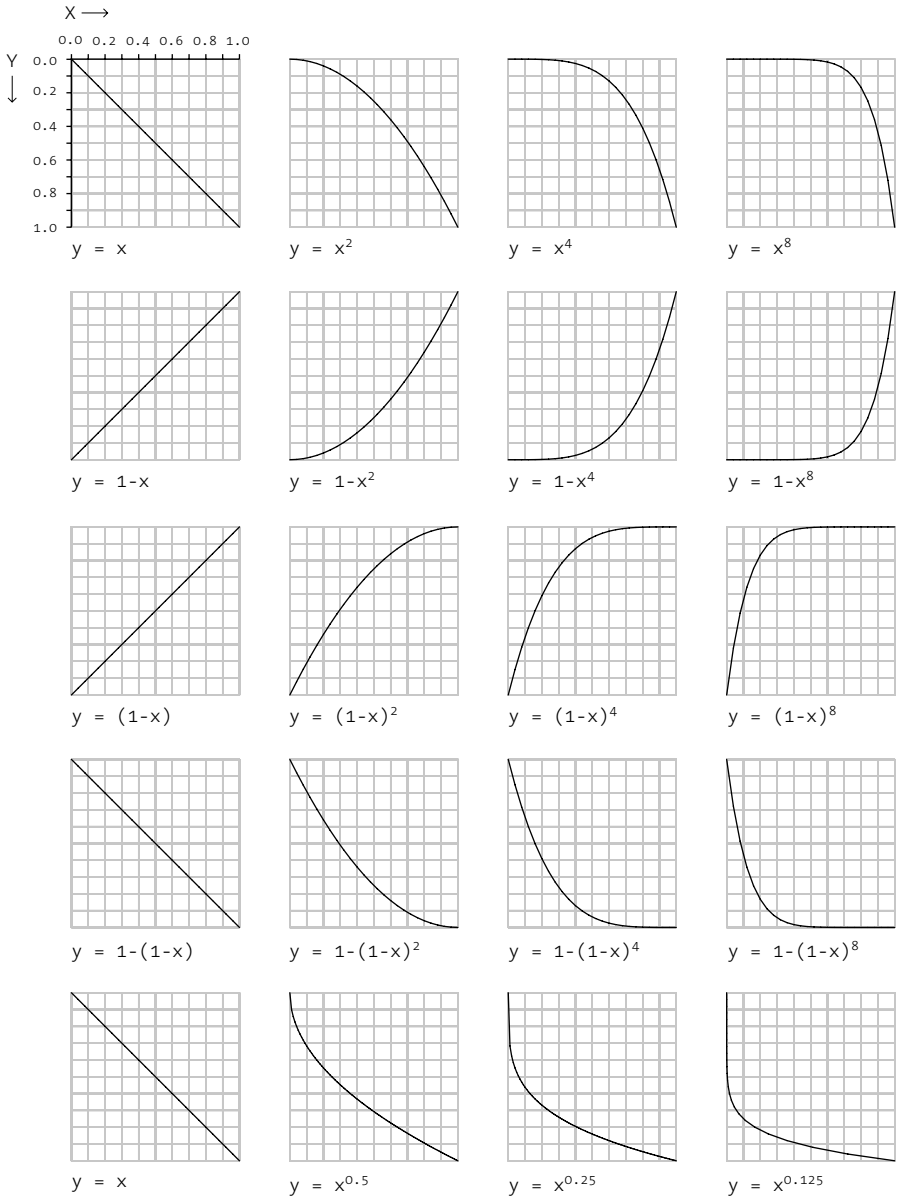
8-06

```
float x = lerp(-20.0, 60.0, 0.0);  // Assign -20.0 to x
float y = lerp(-20.0, 60.0, 0.5);  // Assign 20.0 to y
float z = lerp(-20.0, 60.0, 1.0);  // Assign 60.0 to z
```

The `map()` function is useful to convert directly from one range of numbers to another. It has five parameters.

```
map(value, low1, high1, low2, high2)
```

The *value* parameter is the number to re-map. Similar to the norm function, the *low1* and *low2* parameters are the minimum and maximum values of the number's current range. The *low2* and *high2* parameters are the minimum and maximum values for the new range. The next example shows how to use `map()` to convert values from the range 0 to 255 into the range -1 to 1. This is the same as first normalizing the value, then multiplying and adding to move it from the range 0 to 1 into the range -1 to 1.



Exponential equations

Each of these curves shows the relationship between x and y determined by an equation. The linear equations in the left column are contrasted with exponential curves to the right. Codes 8-08 and 8-09 demonstrate how to translate these curves into code.

```
float x = map(20.0, 0.0, 255.0, -1.0, 1.0);   // Assign -0.84 to x
float y = map(0.0, 0.0, 255.0, -1.0, 1.0);    // Assign -1.0 to y
float z = map(255.0, 0.0, 255.0, -1.0, 1.0);  // Assign 1.0 to z
```
8-07

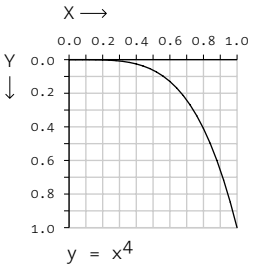
Simple curves

Exponential functions are useful for creating simple curves. Normalized values are used with the `pow()` function to produce exponentially increasing or decreasing numbers that never exceed the value 1. These equations have the form:

$$y = x^n$$

where the value of `x` is between 0.0 and 1.0 and the value of `n` is any integer. In these equations, as the `x` value increases linearly the resulting y value increases exponentially. When continuously plotted, these numbers produce this diagram:

X	Y
0.0	0.0
0.2	0.0016
0.4	0.0256
0.6	0.1296
0.8	0.4096
1.0	1.0



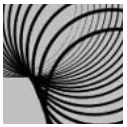
The following example shows how to put this equation into code. It iterates over numbers from 0 to 100 and normalizes the values before making the curve calculation.

```
for (int x = 0; x < 100; x++) {
  float n = norm(x, 0.0, 100.0);  // Range 0.0 to 1.0
  float y = pow(n, 4);            // Calculate curve
  y *= 100;                       // Range 0.0 to 100.0
  point(x, y);
}
```
8-08

Other curves can be created by changing the parameters to `pow()` in line 3.

```
for (int x = 0; x < 100; x++) {
  float n = norm(x, 0.0, 100.0);  // Range 0.0 to 1.0
  float y = pow(n, 0.4);          // Calculate curve
  y *= 100;                       // Range 0.0 to 100.0
  point(x, y);
}
```
8-09

The following three examples demonstrate how the same curve is used to draw different shapes and patterns.



8-10

```
// Draw circles at points along the curve y = x^4
noFill();
smooth();
for (int x = 0; x < 100; x += 5) {
  float n = norm(x, 0.0, 100.0);  // Range 0.0 to 1.0
  float y = pow(n, 4);  // Calculate curve
  y *= 100;  // Scale y to range 0.0 to 100.0
  strokeWeight(n * 5);  // Increase thickness
  ellipse(x, y, 120, 120);
}
```

8-11

```
// Draw a line from the top of the display window to
// points on a curve y = x^4 from x in range -1.0 to 1.0
for (int x = 5; x < 100; x += 5) {
  float n = map(x, 5, 95, -1, 1);
  float p = pow(n, 4);
  float ypos = lerp(20, 80, p);
  line(x, 0, x, ypos);
}
```

8-12

```
// Create a gradient from y = x and y = x^4
for (int x = 0; x < 100; x++) {
  float n = norm(x, 0.0, 100.0);  // Range 0.0 to 1.0
  float val = n * 255.0;
  stroke(val);
  line(x, 0, x, 50);  // Draw top gradient
  float valSquare = pow(n, 4) * 255.0;
  stroke(valSquare);
  line(x, 50, x, 100);  // Draw bottom gradient
}
```

Exponential curves are used in this unit to generate form, but code 23-06 and 31-09 in subsequent units demonstrate their use to control motion and response.

Exercises

1. *Draw the curve $y = 1 - x^4$ to the display window.*
2. *Use the data from the curve $y = x^8$ to draw something unique.*
3. *Compose a range of gradients created from curves.*

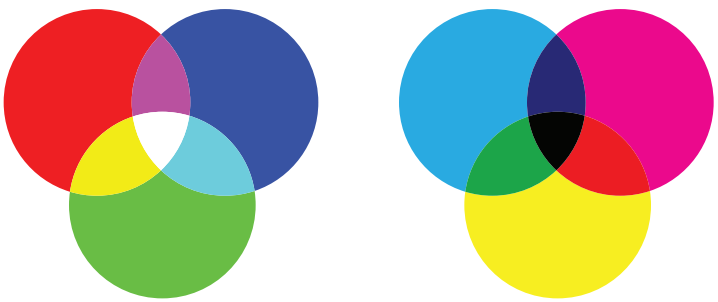
Color 1: Color by Numbers

This unit introduces code elements and concepts for working with color in software.

Syntax introduced:
`color, color(), colorMode()`

When Casey and Ben studied color in school, they spent hours carefully mixing paints and applying it to sheets of paper. They cut paper into perfect squares and carefully arranged them into precise gradations from blue to orange, white to yellow, and many other combinations. Over time, they developed an intuition that allowed them to achieve a specific color value by mixing the appropriate components. Through focused labor, they learned how to isolate properties of color, understand the interactions between colors, and discuss qualities of color.

Working with color on screen is different from working with color on paper or canvas. While the same rigor applies, knowledge of pigments for painting (cadmium red, Prussian blue, burnt umber) and from printing (cyan, yellow, magenta) does not translate into the information needed to create colors for digital displays. For example, adding all the colors together on a computer monitor produces white, while adding all the colors together with paint produces black (or a strange brown). A computer monitor mixes colors with light. The screen is a black surface, and colored light is added. This is known as additive color, in contrast to the subtractive color model for inks on paper and canvas. This image presents the difference between these models:



Additive color

Subtractive color

The most common way to specify color on the computer is with RGB values. An RGB value sets the amount of red, green, and blue light in a single pixel of the screen. If you look closely at a computer monitor or television screen, you will see that each pixel is comprised of three separate light elements of the colors red, green, and blue; but because our eyes can see only a limited amount of detail, the three colors mix to create a single color. The intensities of each color element are usually specified with values between 0 and 255 where 0 is the minimum and 255 is the maximum. Many software applications

also use this range. Setting the red, green, and blue components to 0 creates black. Setting these components to 255 creates white. Setting red to 255 and green and blue to 0 creates an intense red.

Selecting colors with convenient numbers can save effort. For example, it's common to see the parameters (0, 0, 255) used for blue and (0, 255, 0) for green. These combinations are often responsible for the garish coloring associated with technical images produced on the computer. They seem extreme and unnatural because they don't account for the human eye's ability to distinguish subtle values. Colors that appeal to our eyes are usually not convenient numbers. Rather than picking numbers like 0 and 255, try using a color selector and choosing colors. Processing's color selector is opened from the Tools menu. Colors are selected by clicking a location on the color field or by entering numbers directly. For example, in the figure on the facing page, the current blue selected is defined by an R value of 35, a G value of 211, and a B value of 229. These numbers can be used to recreate the chosen color in your code.

Setting colors

In Processing, colors are defined by the parameters to the `background()`, `fill()`, and `stroke()` functions:

```
background(value1, value2, value3)
fill(value1, value2, value3)
fill(value1, value2, value3, alpha)
stroke(value1, value2, value3)
stroke(value1, value2, value3, alpha)
```

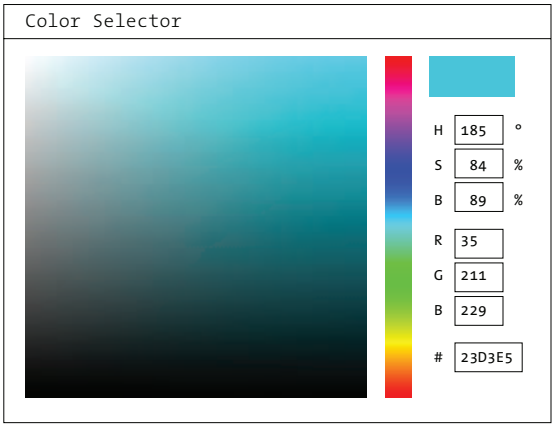
By default, the *value1* parameter defines the red color component, *value2* the green component, and *value3* the blue. The optional alpha parameter to `fill()` or `stroke()` defines the transparency. The *alpha* parameter value 255 means the color is entirely opaque, and the value 0 means it's entirely transparent (it won't be visible).

```
background(242, 204, 47);
```

9-01

```
background(174, 221, 60);
```

9-02



Color Selector

Drag the cursor inside the window or input numbers to select a color. The large square area determines the saturation and brightness, and the thin vertical strip determines the hue. The numeric value of the selected color is displayed in HSB, RGB, and hexadecimal notation.

```
background(129, 130, 87);
noStroke();
fill(174, 221, 60);
rect(17, 17, 66, 66);
```

9-03

```
background(129, 130, 87);
noFill();
strokeWeight(4);
stroke(174, 221, 60);
rect(19, 19, 62, 62);
```

9-04

```
background(116, 193, 206);
noStroke();
fill(129, 130, 87, 102);  // More transparent
rect(20, 20, 30, 60);
fill(129, 130, 87, 204);  // Less transparent
rect(50, 20, 30, 60);
```

9-05

```
background(116, 193, 206);
int x = 0;
noStroke();
for (int i = 51; i <= 255; i += 51) {
  fill(129, 130, 87, i);
  rect(x, 20, 20, 60);
  x += 20;
}
```

9-06

9-07

```
background(56, 90, 94);
smooth();
strokeWeight(12);
stroke(242, 204, 47, 102);  // More transparency
line(30, 20, 50, 80);
stroke(242, 204, 47, 204);  // Less transparency
line(50, 20, 70, 80);
```

9-08

```
background(56, 90, 94);
smooth();
int x = 0;
strokeWeight(12);
for (int i = 51; i <= 255; i += 51) {
  stroke(242, 204, 47, i);
  line(x, 20, x+20, 80);
  x += 20;
}
```

Transparency can be used to create new colors by overlapping shapes. The colors originating from overlaps depend on the order in which the shapes are drawn.

9-09

```
background(0);
noStroke();
smooth();
fill(242, 204, 47, 160);   // Yellow
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160);   // Green
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160);  // Blue
ellipse(57, 79, 64, 64);
```

9-10

```
background(255);
noStroke();
smooth();
fill(242, 204, 47, 160);   // Yellow
ellipse(47, 36, 64, 64);
fill(174, 221, 60, 160);   // Green
ellipse(90, 47, 64, 64);
fill(116, 193, 206, 160);  // Blue
ellipse(57, 79, 64, 64);
```

Color data

The `color` data type is used to store colors in a program, and the `color()` function is used to assign a `color` variable. The `color()` function can create gray values, gray values with transparency, color values, and color values with transparency. Variables of the `color` data type can store all of these configurations:

```
color(gray)
color(gray, alpha)
color(value1, value2, value3)
color(value1, value2, value3, alpha)
```

The parameters of the `color()` function define a color. The *gray* parameter used alone or with *alpha* defines tones ranging from white to black. The *alpha* parameter defines transparency with values ranging from 0 (transparent) to 255 (opaque). The *value1*, *value2*, and *value3* parameters define values for the different components. Variables of the `color` data type are defined and assigned in the same way as the `int` and `float` data types discussed in Data 1 (p. 37).

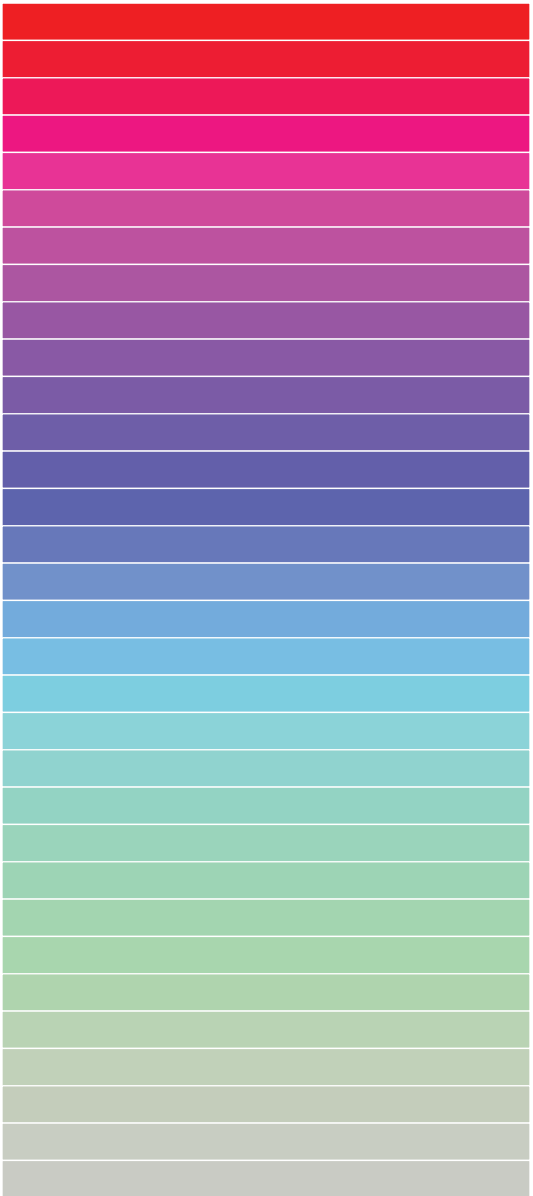
```
color c1 = color(51);                // Creates gray                          9-11
color c2 = color(51, 204);           // Creates gray with transparency
color c3 = color(51, 102, 153);      // Creates blue
color c4 = color(51, 102, 153, 51);  // Creates blue with transparency
```

After a `color` variable has been defined, it can be used as the parameter to the `background()`, `fill()`, and `stroke()` functions.

```
color ruby = color(211, 24, 24, 160);                                        9-12
color pink = color(237, 159, 176);
background(pink);
noStroke();
fill(ruby);
rect(35, 0, 20, 100);
```

RGB, HSB

Processing uses the RGB color model as its default for working with color, but the HSB specification can be used instead to define colors in terms of their hue, saturation, and brightness. The hue of a color is what most people normally think of as the color name: yellow, red, blue, orange, green, violet. A pure hue is an undiluted color at its most intense. The saturation is the degree of purity in a color. It is the continuum from the undiluted, pure hue to its most diluted and dull. The brightness of a color is its relation to light and dark.



RGB			HSB			HEX
255	0	0	360	100	100	#FF0000
252	9	45	351	96	99	#FC0A2E
249	16	85	342	93	98	#F91157
249	23	126	332	90	98	#F91881
246	31	160	323	87	97	#F720A4
244	38	192	314	84	96	#F427C4
244	45	226	304	81	96	#F42EE7
226	51	237	295	78	95	#E235F2
196	58	237	285	75	95	#C43CF2
171	67	234	276	71	94	#AB45EF
148	73	232	267	68	93	#944BED
126	81	232	257	65	93	#7E53ED
108	87	229	248	62	92	#6C59EA
95	95	227	239	59	91	#5F61E8
102	122	227	229	56	91	#667DE8
107	145	224	220	53	90	#6B94E5
114	168	224	210	50	90	#72ACE5
122	186	221	201	46	89	#7ABEE2
127	200	219	192	43	88	#7FCDE0
134	216	219	182	40	88	#86DDE0
139	216	207	173	37	87	#8BDDD4
144	214	195	164	34	86	#90DBC7
151	214	185	154	31	86	#97DBBD
156	211	177	145	28	85	#9CD8B5
162	211	172	135	25	85	#A2D8B0
169	209	169	126	21	84	#A9D6AD
175	206	169	117	18	83	#AFD3AD
185	206	175	107	15	83	#BAD3B3
192	204	180	98	12	82	#C1D1B8
197	201	183	89	9	81	#C5CEBB
202	201	190	79	6	81	#CACEC2
202	200	193	70	3	80	#CACCC5

Color by numbers

Every color within a program is set by numbers, and there are more than 16 million colors to choose from. This diagram presents a few colors and their corresponding numbers for the RGB and HSB color models. The RGB column is in relation to `colorMode(RGB, 255)` *and the HSB column is in relation to* `colorMode(HSB, 360, 100, 100)`.

The `colorMode()` function sets the color space for a program:

```
colorMode(mode)
colorMode(mode, range)
colorMode(mode, range1, range2, range3)
```

The parameters to `colorMode()` change the way Processing interprets color data. The *mode* parameter can be either `RGB` or `HSB`. The range parameters allow Processing to use different values than the default of 0 to 255. A range of values frequently used in computer graphics is between 0.0 and 1.0. Either a single range parameter sets the range for all the color components, or the *range1*, *range2*, and *range3* parameters set the range for each—either red, green, blue or hue, saturation, brightness, depending on the value of the *mode* parameter.

```
// Set the range for the red, green, and blue values from 0.0 to 1.0
colorMode(RGB, 1.0);
```

9-13

A useful setting for `HSB` mode is to set the *range1*, *range2*, and *range3* parameters respectively to 360, 100, and 100. The hue values from 0 to 360 are the degrees around the color wheel, and the saturation and brightness values from 0 to 100 are percentages. This setting matches the values used in many color selectors and therefore makes it easy to transfer color data between other programs and Processing:

```
// Set the range for the hue to values from 0 to 360 and the
// saturation and brightness to values between 0 and 100
colorMode(HSB, 360, 100, 100);
```

9-14

The following examples reveal the differences between hue, saturation, and brightness.

```
// Change the hue, saturation and brightness constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(i*2.5, 255, 255);
  line(i, 0, i, 100);
}
```

9-15

```
// Change the saturation, hue and brightness constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(132, i*2.5, 204);
  line(i, 0, i, 100);
}
```

9-16

```
// Change the brightness, hue and saturation constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(132, 108, i*2.5);
  line(i, 0, i, 100);
}
```

9-17

```
// Change the saturation and brightness, hue constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    stroke(132, j*2.5, i*2.5);
    point(i, j);
  }
}
```

9-18

It's easy to make smooth transitions between colors by changing the values used for *color()*, *fill()*, and *stroke()*. The HSB model has an enormous advantages over the RGB model when working with code because it's more intuitive. Changing the values of the red, green, and blue components often has unexpected results, while estimating the results of changes to hue, saturation, and brightness follows a more logical path. The following examples show a transition from green to blue. The first example makes this transition using the RGB model. It requires calculating all three color values, and the saturation of the color unexpectedly changes in the middle. The second example makes the transition using the HSB model. Only one number needs to be altered, and the hue changes smoothly and independently from the other color properties.

```
// Shift from blue to green in RGB mode
colorMode(RGB);
for (int i = 0; i < 100; i++) {
  float r = 61 + (i*0.92);
  float g = 156 + (i*0.48);
  float b = 204 - (i*1.43);
  stroke(r, g, b);
  line(i, 0, i, 100);
}
```

9-19

```
// Shift from blue to green in HSB mode
colorMode(HSB, 360, 100, 100);
for (int i = 0; i < 100; i++) {
  float newHue = 200 - (i*1.2);
  stroke(newHue, 70, 80);
  line(i, 0, i, 100);
}
```

9-20

Hexadecimal

Hexadecimal (hex) notation is an alternative notation for defining color. This method is popular with designers working on the Web because standards such as HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) use this notation. Hex notation for color encodes each of the numbers from 0 to 255 into a two-digit value using the numbers 0 through 9 and the letters *A* through *F*. In this way three RGB values from 0 to 255 can be written as a single six-digit hex value. A few sample conversions demonstrate this notation:

RGB	Hex
255, 255, 255	#FFFFFF
0, 0, 0	#000000
102, 153, 204	#6699CC
195, 244, 59	#C3F43B
116, 206, 206	#74CECE

Converting color values from RGB to hex notation is not intuitive. Most often, the value is taken from a color selector. For instance, you can copy and paste a hex value from Processing's color selector into your code. When using color values encoded in hex notation, you must place a # before the value to distinguish it within the code.

9-21

```
// Code 9-03 rewritten using hex numbers
background(#818257);
noStroke();
fill(#AEDD3C);
rect(17, 17, 66, 66);
```

There's more information about hex notation in Appendix D (p. 669).

Exercises

1. *Explore a wide range of color combinations within one composition.*
2. *Use HSB color and a* `for` *structure to design a gradient between two colors.*
3. *Redraw your composition from exercise 1 using hexadecimal color values.*

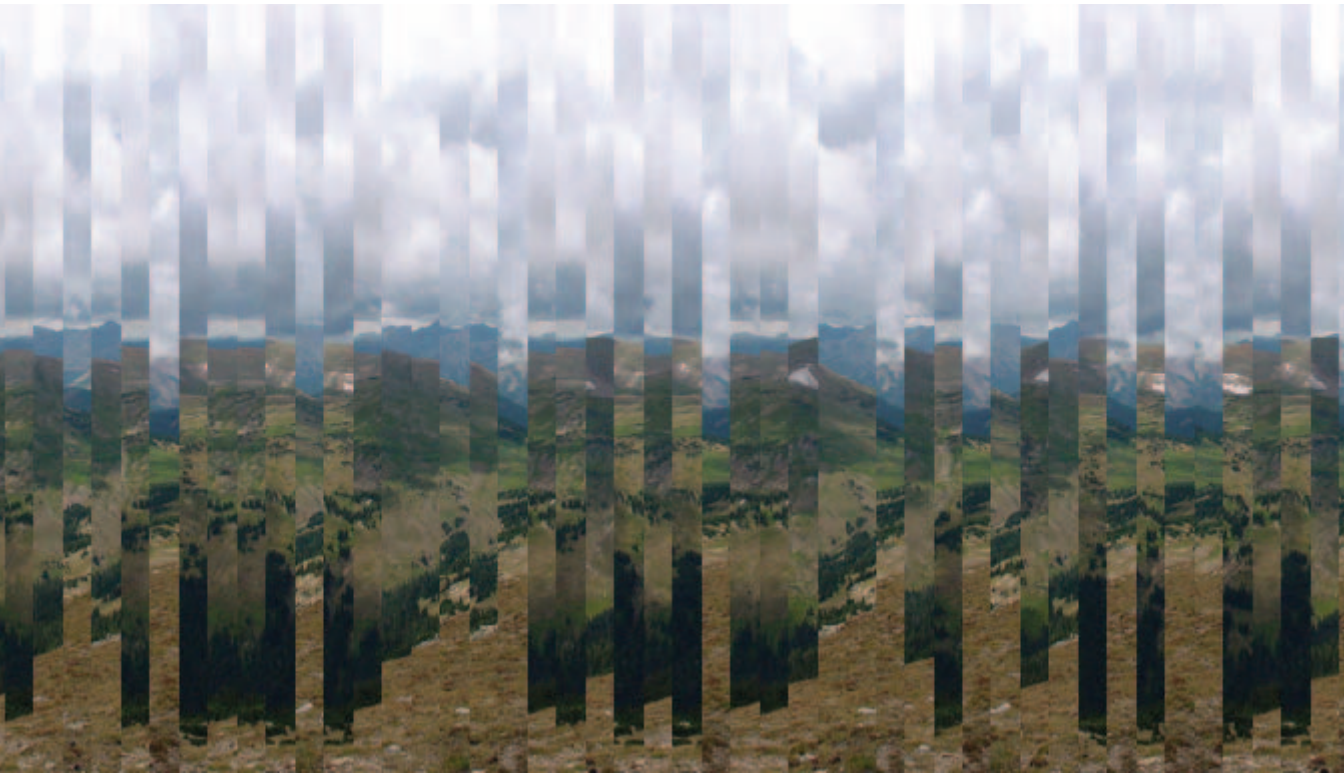


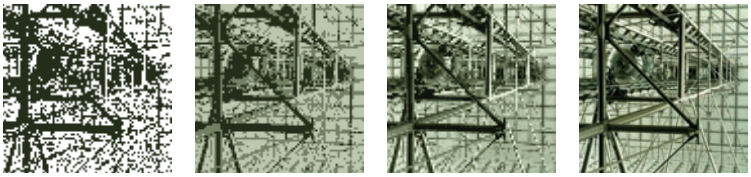
Image 1: Display, Tint

This unit introduces loading and displaying images.

Syntax introduced:

```
PImage, loadImage(), image()
tint(), noTint()
```

Digital photographs are fundamentally different from analog photographs captured on film. Like computer screens, digital photos are rectangular grids of color. The dimensions of digital images are measured in units of pixels. If an image is 320 pixels wide and 240 pixels high, it has 76,800 total pixels. If an image is 1280 pixels wide and 1024 pixels wide, the total number of pixels is an impressive 1,310,720 (1.3 megapixels). Every digital image has a color depth. The color depth refers to the number of bits (p. 669) used to store each pixel. If the color depth of an image is 1, each pixel can be one of two values, for example, black or white. If the color depth is 4, each pixel can be one of 16 values. If the color depth of an image is 8, each pixel can be one of 256 values. Looking at the same image displayed with different color depths reveals how this affects the image's appearance:



1-bit (1 color) 2-bit (4 colors) 4-bit (16 colors) 8-bit (256 colors)

When the Apple Macintosh computer was introduced in 1984, it had a black-and-white screen. Since then, the reproduction of color on screen has rapidly improved. Many contemporary screens have a color depth of 24, which means each pixel can be one of 16,777,216 available colors. This number is typically referred to as "millions of colors."

Digital images are comprised of numbers representing colors. The file format of an image determines how the numbers are ordered in the file. Some file formats store the color data in mathematically complex arrangements to compress the data and reduce the size of the resulting file. A program that loads an image file must know the file format of the image so it can translate the file's data into the expected image. Different types of digital image formats serve specific needs. Processing can load GIF, JPEG, and PNG images, along with some other formats as described in the reference. If you don't already have your images in one of these formats, you can convert other types of digital images to these formats with programs such as GIMP or Adobe Photoshop. Refer to the documentation for these programs if you're unsure how to convert images.

How do you know which image format to use? They all have obscure names that don't help in making this decision, but each format's advantages becomes clear through comparison:

Format	Extension	Color depth	Transparency
GIF	.gif	1-bit to 8-bit	1-bit
JPEG	.jpg	24-bit	None
PNG	.png	1-bit to 24-bit	8-bit

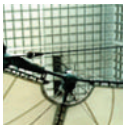
If you are displaying your work on the Internet, image compression becomes an important issue. GIF images are useful for simple graphics with a limited number of colors and transparency. PNG images have similar characteristics but support the full range of colors and transparency. The JPEG format works well for photos, and JPEG files will be smaller than most images saved as PNG. This is because JPEG is a "lossy" format, which means it sacrifices some image quality to reduce file size.

Display

Processing can load images, display them on the screen, and change their size, position, opacity, and tint. There's a data type for images called `PImage`. The same way that integers are stored in variables of the `int` data type and values of `true` and `false` are stored in the `boolean` data type, images are stored in variables of the `PImage` data type. Before displaying an image, it's necessary to first load it with the `loadImage()` function. Be sure to include the file format extension as a part of the name and to put the entire name in quotes (e.g., "*pup.gif*", "*kat.jpg*", "*ignatz.png*"). For the image to load, it must be in the data folder of the current program. Add the image by selecting the "Add File" option in the Sketch menu of the Processing environment. Navigate to the image's location on your computer, select the image's icon or name, and click "Open" to add it to the sketch's data folder. As a shortcut, you can also drag and drop an image to the Processing window. To make sure the image was added, select "Show Sketch Folder" from the Sketch menu. The image will be inside the *data* folder. With the image file in the right place, you can load and then display it with the `image()` function:

```
image(name, x, y)
image(name, x, y, width, height)
```

The parameters for `image()` determine the image to draw and its position and size. The *`name`* parameter must be a `PImage` variable. The *`x`* and *`y`* parameters set the position of the upper-left corner of the image. The image will display at its actual size (in units of pixels), but you can change the size by adding the *`width`* and *`height`* parameters. Be careful to use the correct capitalization when loading images. If the image is *arch.jpg*, trying to load *Arch.jpg* or *arch.JPG* will create an error. Also, avoid the use of spaces in image names, which can cause problems.



10-01

```
PImage img;
// Image must be in the sketch's "data" folder
img = loadImage("arch.jpg");
image(img, 0, 0);
```

10-02

```
PImage img;
// Image must be in the sketch's "data" folder
img = loadImage("arch.jpg");
image(img, 20, 20, 60, 60);
```

Image color, Transparency

Images are colored with the `tint()` function. This function is used the same way as `fill()` and `stroke()`, but it affects only images:

```
tint(gray)
tint(gray, alpha)
tint(value1, value2, value3)
tint(value1, value2, value3, alpha)
tint(color)
```

All images drawn after running `tint()` will be tinted by the color specified in the parameters. This has no permanent effect on the images, and running the `noTint()` function disables the coloration for all images drawn after it is run.

10-03

```
PImage img;
img = loadImage("arch.jpg");
tint(102); // Tint gray
image(img, 0, 0);
noTint(); // Disable tint
image(img, 50, 0);
```

10-04

```
PImage img;
img = loadImage("arch.jpg");
tint(0, 153, 204); // Tint blue
image(img, 0, 0);
noTint(); // Disable tint
image(img, 50, 0);
```

10-05

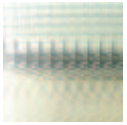
```
color yellow = color(220, 214, 41);
color green = color(110, 164, 32);
color tan = color(180, 177, 132);
PImage img;
img = loadImage("arch.jpg");
tint(yellow);
image(img, 0, 0);
tint(green);
image(img, 33, 0);
tint(tan);
image(img, 66, 0);
```

The parameters for `tint()` follow the color space determined by the `colorMode()` function (remember, the default color mode is RGB, with all values ranging from 0 to 255). If the color mode is changed to HSB or a different range, the tint values should be specified relative to that mode.

To make an image transparent without changing its color, set the tint to white. The value will depend on the current color mode, but the default white value is 255.

10-06

```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 102);  // Alpha to 102 without changing the tint
image(img, 0, 0, 100, 100);
tint(255, 204, 0, 153);  // Tint to yellow, alpha to 153
image(img, 20, 20, 100, 100);
```



10-07

```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 51);
// Draw the image 10 times, moving each to the right
for (int i = 0; i < 10; i++) {
  image(img, i*10, 0);
}
```

GIF and PNG images retain their transparency when loaded and displayed in Processing. This allows anything drawn before the image to be visible through the transparent sections of the image. GIF images have only 1-bit transparency, meaning each pixel can only be completely opaque or completely transparent. The PNG format supports 8-bit transparency, meaning there are 256 levels of opacity.

10-08

```
// Loads a GIF image with 1-bit transparency
PImage img;
img = loadImage("archTrans.gif");
background(255);
image(img, 0, 0);
image(img, -20, 0);
```

10-09

```
// Loads a PNG image with 8-bit transparency
PImage img;
img = loadImage("arch.png");
background(255);
image(img, 0, 0);
image(img, -20, 0);
```

Exercises

1. *Draw two images in the display window.*
2. *Draw three images in the display window, each with a different tint.*
3. *Load a GIF or PNG image with transparency and create a collage by layering the image.*

37	%	56	8	75	K
38	&	57	9	76	L
39	′	58	:	77	M
40	(	59	;	78	N
41	)	60	<	79	O
42	*	61	=	80	P
43	+	62	>	81	Q
44	,	63	?	82	R
45	-	64	@	83	S
46	.	65	A	84	T
47	/	66	B	85	U
48	0	67	C	86	V
49	1	68	D	87	W
50	2	69	E	88	X
51	3	70	F	89	Y
52	4	71	G	90	Z
53	5	72	H	91	[
54	6	73	I	92	\

Data 2: Text

This unit introduces code elements for working with language.

Syntax introduced:
`char, String`

I SENSE THE SUN IN THE STREET,
ALL SPACE IN THE STREET.
BANG! THE SUN HAS SLID.

This poem was generated from software written by Margaret Masterman and was featured in the 1968 Cybernetic Serendipity exhibition at the Institute of Contemporary Arts (ICA) in London. This exhibition exposed the public to examples of software-generated poems, music, and drawings. While the poem may or may not conform to your ideas about great poetry, the exhibition was important for its early emphasis on using the computer as a language processing machine. A common misconception holds that computer programming is applicable only to technical fields. While there is a strong connection between programming and technology, it's not the only realm in which computers can make for interesting collaborators. Programming can be approached with an emphasis on language, making computers potentially interesting to a far broader audience.

Some of the earliest explorations of the computer outside scientific research focused on software as a language engine. The history of artificial intelligence (AI) has a strong component of language processing. John McCarthy's LISP programming language made processing text easy and became popular for early experimentation in AI. The controversial ELIZA software, written by Joseph Weizenbaum in 1966, parodies the dialog between a Rogerian therapist and a patient by rephrasing the patient's statements as questions. People input statements through a keyboard and the software constructs a reply. For example, if the patient types "I feel depressed," ELIZA might respond, "Why do you say you are depressed?" Terry Winograd's SHRDLU project, c. 1970, used the same kind of interaction between keyboard input and text response, but it earnestly explored the computer's potential for understanding natural language. SHRDLU made it possible for a person to have a discussion with the computer about an arrangement of simulated blocks. For example, to the query "How many blocks are not in the box?" the software would respond "Four of them" based on the current status of the blocks.

Researchers have continued to explore language as an interface with and input to software. Emerging software services such as automated translation and speech-to-text conversions are not always reliable, but they are fascinating to explore. For example, if we take two simple English sentences . . .

Translation requires nuance. Can it be performed by a machine?

. . . and convert them to Italian using an online translation service, we are given this text:

La traduzione richiede la sfumatura. Può essere effettuata da una macchina?

If we take the Italian translation and convert it back to English, we now have

The translation demands the shading. Can be carried out from one machine?

Similarly, software designed to convert spoken language into written language has its limitations. Both technologies, however, can be used in controlled circumstances as unique ways of working with text and software.

This unit does not discuss artificial intelligence or language parsing, but text is one of the most common types of data created and modified by software. The text created for Email, publications, and Web pages is a vast resource of data that can be stored and presented through the data types and functions introduced below.

Characters

The `char` data type stores typographic symbols such as *A*, *d*, *5*, and *$*. The name *char* is short for *character*, and this type of data is distinguished from other typographic symbols in the program by surrounding single quotes. `Char` variables are declared and assigned in the same way as the `int` and `float` types.

```
char a = 'n';      // Assign 'n' to variable a                    11-01
char b = n;        // ERROR! Without quotes, n is a variable
char c = "n";      // ERROR! The "" defines n as a String, not a char
char d = 'not';    // ERROR! The char type can hold only one character
```

The following example creates a new `char` variable, assigns values, and prints the values to the console.

```
char letter = 'A'; // Declare variable letter and assign 'A'        11-02
println(letter);   // Prints "A" to the console
letter = 'B';      // Assign 'B' to variable letter
println(letter);   // Prints "B" to the console
```

Many characters have a corresponding number on the standardized ASCII table. For example, *A* is 65, *B* is 66, *C* is 67, etc. You can find which character matches which number by looking at an ASCII table (such as in Appendix C, p. 664) or by testing with the `println()` function:

```
char letter = 'A'; // Declare variable letter and assign 'A'        11-03
println(letter);   // Prints "A" to the console
int n = letter;    // Assign the numerical value of 'A' to variable n
println(n);        // Prints "65" to the console
```

Appendix C also includes information about using non-ASCII characters (for instance, a character with an accent or an umlaut) or characters from non-Roman alphabets such as Japanese or Korean.

The mapping between numeric and alphabetic formats emphasizes the importance of data types. The following program prints the letters *A* to *Z* to the console by incrementing the `char` variable in a `for` structure.

```
char letter = 'A';  // Declare variable letter and assign 'A'
for (int i = 0; i < 26; i++) {
  print(letter);    // Prints a character to the console
  letter++;         // Add 1 to the value of the character
}
println('.');       // Adds a period to the end of the alphabet
```
11-04

Words, Sentences

Use the `String` data type to store words and sentences. Surrounding double quotes distinguish strings from characters and the rest of the program. Quotation marks define "s" as a string, while single quotes (apostrophes) define 's' as a character, and without either it could be a variable name. The `String` data type is different from the data types `int`, `float`, and `char` because it is an *object* (p. 395), a composite data type containing multiple data elements and functions. The previously introduced data types `PImage` and `PFont` are also objects. `String` variables are declared and assigned in the familiar way, but the word `String` must be capitalized:

```
String a = "Eponymous";  // Assign "Eponymous" to a
String b = 'E';          // ERROR! The '' define E as a char
String c = "E";          // Assign "E" to c
string d = "E";          // ERROR! String must be capitalized
```
11-05

The following example demonstrates some basic ways to use this data type:

```
// The String data type can contain long and short text elements
String s1 = "Rakete bee bee?";
String s2 = "Rrrrrrrrrrrrrrrrummmmmpffff tillffff tooooo?";
println(s1);  // Prints "Rakete bee bee?"
println(s2);  // Prints "Rrrrrrrrrrrrrrrrummmmmpffff tillffff tooooo?"

// Strings can be combined with the + operator
String s3 = "Rakete ";
String s4 = "rinnzekete";
String s5 = s3 + s4;
println(s5);  // Prints "Rakete rinnzekete"
```
11-06

If you have a large quantity of text to display in your programs, it's better to load the text into the program from a file than to store it in `String` variables. This process is explained in Input 6 (p. 427). The `char` and `String` data types will be used to more interesting ends in the proceeding units on Typography and Input. There are many functions inside the `String` data type for operating on text. They perform actions such as making all the letters lowercase or looking only at one letter within the text. These functions are explained in the next unit.

Exercises

1. *Create five `char` variables and assign a character to each. Write each to the console.*
2. *Create two `String` variables and assign a word to each. Write each to the console.*
3. *Store a sentence in a `String` and write it to the console.*

Data 3: Conversion, Objects

This unit introduces converting values from one data format to another and working with data as objects.

Syntax introduced:
`boolean(), byte(), char(), int(), float(), str()`
`“.” (dot operator)`
`PImage.width, PImage.height`
`String.length(), String.startsWidth(), String.endsWidth(),`
`String.charAt(), String.toCharArray(), String.subString(),`
`String.toLowerCase(), String.toUpperCase()`
`String.equals()`

When a variable is created, its data type is specified. If the variable will store numeric data, the `int` or `float` types are used. If the variable will store character data, a `String` can be used to store multiple characters, or the `char` data type can store just one. A `true` or `false` value is stored in a `boolean` variable, an image is stored in a `PImage` variable, and a typeface is stored in a `PFont` font variable. After the variable is created, it can only be assigned data elements of its type. Sometimes, though, it's necessary to convert a value from one type of data to another, a task for which Processing has several functions.

The data types `int`, `float`, `boolean`, and `char` are called *primitive* data types because they store a single data element. The types `String`, `PImage`, and `PFont` are different. Variables created from these data types are *objects*. Objects are usually composed of several primitive data types (or other objects), and can also have functions inside to act on their data. For example, a `String` object stores an array of characters and has functions that return the number of characters or the character at a specific location. Objects are visually distinguished from primitive data types with capitalization.

Data conversion

Some data type conversions are automatic and others need to be made explicit with functions written for data type conversion. Automatic conversions are made between compatible types. For example, an `int` can be automatically converted to a `float`, but a `float` can't be automatically converted to an `int`:

12-01
```
float f = 12.6;
int i = 127;
f = i;  // Converts 127 to 127.0
i = f;  // Error: Can't automatically convert a float to an int
```

How does one know which data types are compatible and which require an explicit conversion? Conversions that involve a loss of information must be explicit. When converting an `int` to a `float`, nothing is lost. When converting a `float` to an `int`, however, the numbers after the decimal point are lost. Explicit conversions are a way of stating in code that this loss of information is intentional. The functions for explicit data type conversion are `boolean()`, `byte()`, `char()`, `float()`, `int()`, and `str()`. Each is used to convert other data types to the type for which the function is named.

The `boolean()` function converts the number 0 to `false` and all other numbers to `true`. It converts the string "true" to `true` and the string "false" to `false`.

12-02

```
int i = 0;
boolean b = boolean(i);  // Assign false to b
int n = 12;
b = boolean(n);          // Assign true to b
String s = "false";
b = boolean(s);          // Assign false to b
```

The `byte()` function converts other types of data to a byte representation. A `byte` can only be a whole number between –128 and 127; therefore, when a number outside this range is converted, its value wraps to the corresponding byte representation.

12-03

```
float f = 65.0;
byte b = byte(f);  // Assign 65 to b
char c = 'E';
b = byte(c);       // Assign 69 to b
f = 130.0;
b = byte(f);       // Assign -126 to b
```

The `char()` function converts other types of data to a character representation. An explanation of the numbering can be found in Appendix C (p. 664).

12-04

```
int i = 65;
byte y = 72.0;
char c = char(i);  // Assign 'A' to c
c = char(y);       // Assign 'H' to c
```

The `float()` function converts other types of data to a floating-point representation. It is most often used when making calculations. As discussed in Math 1 (p. 43), dividing two integers will always evaluate as an integer, which is a problem when working with fractions. For example, when the integer number 3 is divided by 6, the answer is the integer value 0 rather than the often desired floating-point value 0.5. Converting one of these values to a `float` allows the expression to evaluate to a floating-point value.

```
int i = 2;                                                    12-05
int j = 3;
float f1 = i/j;          // Assign 0.0 to f1
float f2 = i/float(j);   // Assign 0.6666667 to f2
```

The `int()` function converts other types of data to an integer representation. Many of the math functions only return `float` values, and it's necessary to convert them to integers for use in other parts of a program.

```
float f = 65.3;                                               12-06
int i = int(f);  // Assign 65 to i
char c = 'E';
i = int(c);      // Assign 69 to i
```

The `str()` function converts other types of data to a string representation:

```
int i = 3;                                                    12-07
String s = str(i);  // Assign "3" to s
float f = -12.6;
s = str(f);         // Assign "-12.6" to s
boolean b = true;
s = str(b);         // Assign "true" to s
```

The `nf()` function (p. 422) provides more control when converting an `int` or a `float` to a `String`. It can set the number of decimal places and pad the number with zeros.

Objects

Variables created with the `PImage`, `PFont`, and `String` data types are *objects*. Variables within an object are called *fields*, and functions within an object are called *methods*. Fields and methods are accessed with the *dot operator*, a period placed between the name of the object and the name of a data element or function inside the object. The `PImage`, `PFont`, and `String` data types each have their own unique additional data elements and functions.

The `PImage` data type has two fields that store the width and height of the image, named, appropriately, `width` and `height`. To access these, write the name of the object followed by a *dot* and the name of the variable:

```
PImage img = loadImage("ohio.jpg");  // Load a 320 x 240 pixel image   12-08
int w = img.width;   // Assign 320 to w
int h = img.height;  // Assign 240 to h
println(w);  // Prints "320"
println(h);  // Prints "240"
```

The `width` and `height` variables can only be read—assigning a value to them will cause problems. The image's `width` and `height` values can be used to position images side by side or to place shapes in relation to an image. The `PImage` object has many functions for manipulating the pixels of an image. They are discussed in Image 3 (p. 321), Image 4 (p. 347), and Image 5 (p. 355).

The `String` data type includes methods for examining individual characters within the string, extracting parts of strings, converting an entire string to uppercase or lowercase characters, and comparing two `String` variables. Some of the most common `String` methods are introduced below, and more are discussed in the Processing reference.

The `length()` method returns the number of characters in a `String` object:

```
String s1 = "Player Piano";                                    12-09
String s2 = "P";
println(s1.length());  // Prints "12"
println(s2.length());  // Prints "1"
```

Notice the difference in syntax between the array field `length` (p. 304) and the `String` method `length()`. They both calculate the number of elements in their object, but because the technique for getting the number of elements in a `String` is a method, the parentheses are necessary.

The `startsWith()` and `endsWith()` methods test whether a string starts or ends with the string used as the parameter:

```
String s1 = "Slaughterhouse Five";                             12-10
println(s1.startsWith("S"));     // Prints "true"
println(s1.startsWith("Five"));  // Prints "false"
println(s1.endsWith("Five"));    // Prints "true"
```

The `charAt()` method is used to read a single character within a string. This method has one parameter to define the character that is returned.

```
String s = "Verde";                                            12-11
println(s.charAt(0));  // Prints "V"
println(s.charAt(2));  // Prints "r"
println(s.charAt(4));  // Prints "e"
```

The `toCharArray()` method creates an array of characters from the contents of a string.

```
String s = "Azzurro";                                          12-12
char[] c = s.toCharArray();
println(c[0]);  // Prints "A"
println(c[1]);  // Prints "z"
```

The `String` method `substring()` returns a new string that is a part of the original. When the method is used with one parameter, the string is read from the position given as the parameter to the end of the string. When two parameters are used, the string between the two parameter positions is returned.

12-13
```
String s = "Giallo";
println(s.substring(2));     // Prints "allo"
println(s.substring(4));     // Prints "lo"
println(s.substring(1, 4));  // Prints "ial"
println(s.substring(0, s.length()-1));  // Prints "Giall"
```

The `String` method `toLowerCase()` returns a copy of the string with all of the characters made lowercase. The method `toUpperCase()` does the same for uppercase.

12-14
```
String s = "Nero";
println(s.toLowerCase());  // Prints "nero"
println(s.toUpperCase());  // Prints "NERO"
```

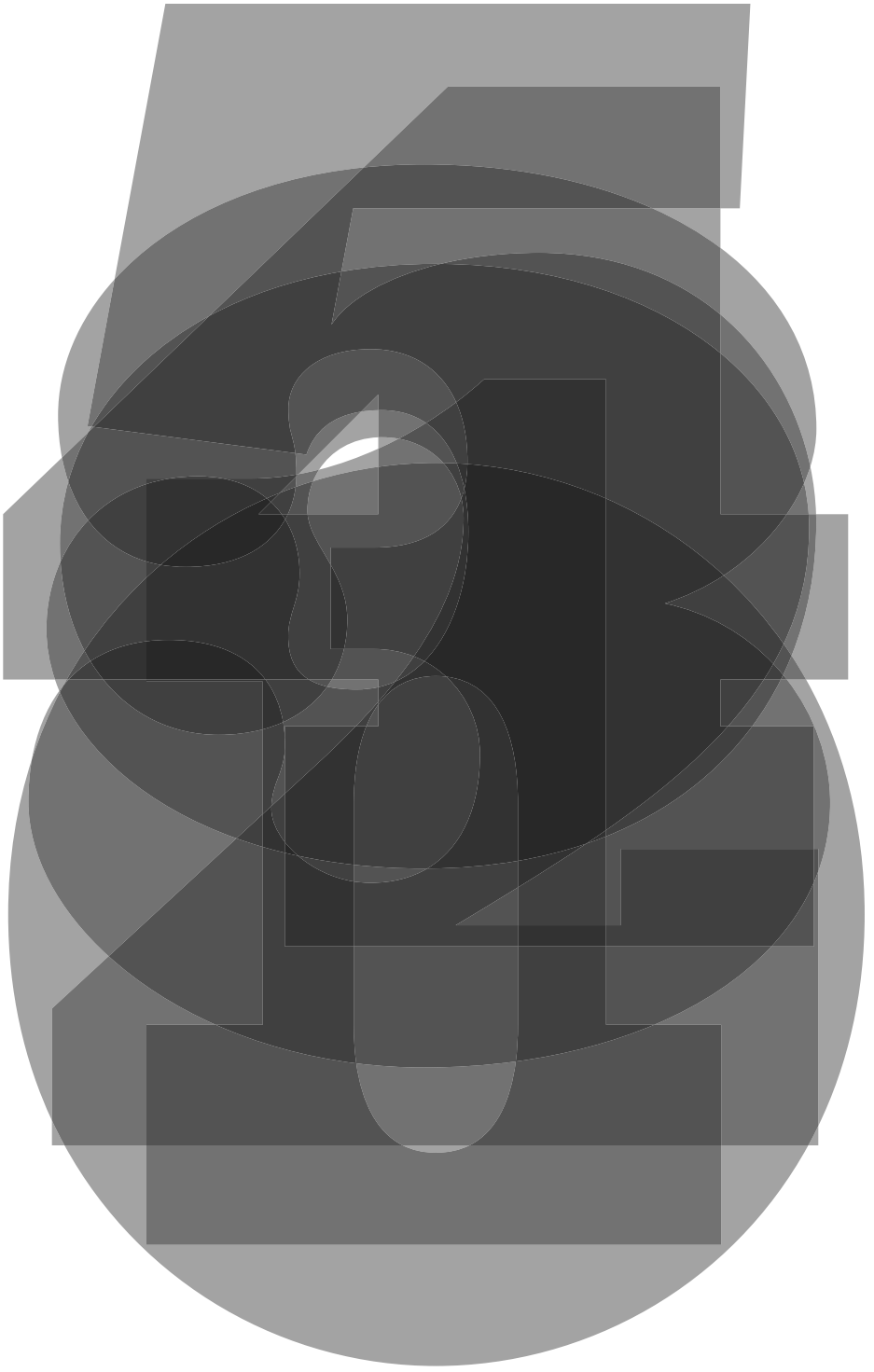
Because the `String` data type is an object, it's not possible to compare two strings with relational operators. Using == to compare two objects will compare only whether they are stored in the same location in memory, not their actual contents. Instead, the `equals()` method is used to determine whether two `String` variables contain the same characters.

12-15
```
String s1 = "Bianco";
String s2 = "Bianco";
String s3 = "Nero";
println(s1.equals(s2));  // Prints "true"
println(s1.equals(s3));  // Prints "false"
```

This unit introduced ways to utilize the additional data and functions within objects. These methods will be useful in the following units, but the full potential of working with objects is not revealed until Structure 4 (p. 395), which presents the techniques for writing your own objects.

Exercises

1. *Write a program to convert the value of an integer to other data types. Display the conversions in the console.*
2. *Load an image and display its height and width to the console using the `PImage` fields.*
3. *Explore the `String` methods and use one or more of them to reconfigure two sentences into one variable.*



Typography 1: Display

This unit introduces loading and setting fonts and displaying letters on screen.

Syntax introduced:

```
PFont, loadFont(), textFont(), text()
textSize(), textLeading(), textAlign(), textWidth()
```

The evolution of typographic reproduction and display technologies has and continues to impact human culture. Early printing techniques developed by Johannes Gutenberg in fifteenth-century Germany using positionable letters cast from lead provided a catalyst for increased literacy and the scientific revolution. Automated typesetting machines, such as the Linotype invented in the nineteenth century, changed the way information was produced, distributed, and consumed. In the digital era, the way we consume text has changed drastically since the proliferation of personal computers in the 1980s and the rapid growth of the Internet in the 1990s. Text from Emails, websites, and instant messages fill computer screens, and while many of the typographic rules of the past apply, type on screen requires additional considerations for the sake of communication and legibility.

Letters on screen are created by setting the color of pixels. The quality of the typography is constrained by the resolution of the screen. Because screens have a low resolution in comparison to paper, techniques have been developed to enhance the appearance of type on screen. The fonts on the earliest Apple Macintosh computers were comprised of small bitmap images created at specific sizes like 10, 12, and 24 points. Using this technology, a variation of each font was designed for each size of a particular typeface. For example, the character *A* in the San Francisco typeface used a different image to display the character at size 12 and 18. When the LaserWriter printer was introduced in 1985, Postscript technology defined fonts with a mathematical description of each character's outline. This allowed type on screen to scale to large sizes and still look smooth. Apple and Microsoft later developed TrueType, another outline font format. More recently, these technologies were merged into the OpenType format. In the meantime, methods to smooth black-and-white text on screen were introduced. These anti-aliasing techniques use gray pixels at the edge of characters to compensate for low screen resolution.

The proliferation of personal computers in the mid-1980s spawned a period of rapid typographic experimentation. Digital typefaces are software, and the old rules of metal and photo type no longer apply. The Dutch typographers known as LettError explain, "The industrial methods of producing typography meant that all letters had to be identical . . . Typography is now produced with sophisticated equipment that doesn't impose such rules. The only limitations are in our expectations."[1] LettError expanded the possibilities of typography with their typeface Beowolf (p. 169). It prints every letter differently so that each time an *A* is printed, for example, it will have a different

shape. During this time, typographers such as Zuzana Licko and Barry Deck began creating radical and innovative typefaces with the assistance of new software tools. The flexibility of software has enabled extensive font revivals and historic homages such as Adobe Garamond from Robert Slimbach and The Proteus Project from Jonathan Hoefler. Typographic nuances such as ligatures—connections between letter pairs such as fi, ff, and æ—made impractical by modern mechanized typography are flourishing again through software font tools.

Loading fonts, Drawing text

Before letters can be displayed on the screen with Processing, a font must first be converted into the VLW format. To convert a font, select the "Create Font" option from the Tools menu. A window opens and displays the names of the fonts installed on your computer that can be converted. Select a font from the list and click "OK." The font generates and is copied into the current sketch's *data* folder. To make sure the font is there, click on the Sketch menu and select "Show Sketch Folder."

Like the early Macintosh fonts, the VLW format used by Processing stores each letter of the alphabet as an image. The VLW format is a quick way to render text and makes it possible to include a font with a sketch. A font created at size 12 will therefore have a smaller file size than a font stored at size 96 because the images require less space. The Create Font dialog box offers the option to set the size of the font and to select whether it will be smooth (antialiased). This box also offers the option to export "All Characters," which means every character in the font will be included. The name of the file can also be changed before the font is created.

After the font is created, drawing letters to the display window is a multistep process. Before a font is used in a program, it must be loaded and set as the current font. Processing has a unique data type called `PFont` to store font data. Make a new variable of the type `PFont` and use the `loadFont()` function to load the font. The `textFont()` function must be used to set the current font. The `text()` function is used to draw characters to the screen:

```
text(data, x, y)
text(stringdata, x, y, width, height)
```

The *`data`* parameter can be a `String`, `char`, `int`, or `float`. The *`stringdata`* parameter can only be a `String`. The *`x`* and *`y`* parameters set the position of the lower-left corner. The optional *`width`* and *`height`* parameters set boundaries. The `text()` function draws the characters at the current font's original size. The `fill()` function controls the color and transparency of text. This function affects text the same way it affects shapes such as `rect()` and `ellipse()`. Text is not affected by `stroke()`.

The following examples use a font named Ziggurat. To run these examples, you will need to use the "Create Font" tool to create your own font. Change the name of the parameter of `loadFont()` to the name of the font that you created.

13-01

```
PFont font;  // Declare the variable
font = loadFont("Ziggurat-32.vlw");  //  Load the font
textFont(font);  // Set the current text font
fill(0);
text("LAX", 0, 40);   // Write "LAX" at coordinate (0,40)
text("AMS", 0, 70);   // Write "AMS" at coordinate (0,70)
text("FRA", 0, 100);  // Write "FRA" at coordinate (0,100)
```

13-02

```
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
text(19, 0, 36);    // Write 19 at coordinate (0,36)
text(72, 0, 70);    // Write 72 at coordinate (0,70)
text('R', 62, 70);  // Write 'R' at coordinate (62,70)
```

13-03

```
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
fill(0);
String s = "Response is the medium";
text(s, 10, 20, 80, 50);
```

13-04

```
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(255);  // White
text("DAY", 0, 40);
fill(0);    // Black
text("CVG", 0, 70);
fill(102);  // Gray
text("ATL", 0, 100);
```

13-05

```
PFont font;
font = loadFont("Ziggurat-72.vlw");
textFont(font);
fill(0, 160);  // Black with low opacity
text("1",  0, 80);
text("2", 15, 80);
text("3", 30, 80);
text("4", 45, 80);
text("5", 60, 80);
```

To use two fonts in one program, create two `PFont` variables and use the `textFont()` function to change the current font.

13-06

```
PFont font1, font2;
font1 = loadFont("Ziggurat-32.vlw");
font2 = loadFont("ZigguratItalic-32.vlw");
fill(0);
// Set the font to Ziggurat-32.vlw
textFont(font1);
text("GNU", 6, 45);
// Set the font to ZigguratItalic-32.vlw
textFont(font2);
text("GNU", 2, 80);
```

Text attributes

Processing includes functions to control the way text is displayed—for example, by changing its size, leading (the spacing between lines), and alignment. Processing can also calculate the width of any character or group of characters, a useful function for arranging shapes and typographic elements.

Fonts in Processing are images and not vector outlines. When the font is drawn at a different size from the size at which it was created, it is scaled and therefore does not always look as crisp and smooth. For example, if a font is created at 12 pixels and is displayed at 96 pixels, it will appear blurry. The `textSize()` function sets the current font size:

```
textSize(size)
```

The *`size`* parameter defines the dimension of the letters in units of pixels.

13-07

```
// Reducing a font created at 32 pixels
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
text("LNZ", 0, 40);   // Large
textSize(18);
text("STN", 0, 75);   // Medium
textSize(12);
text("BOS", 0, 100);  // Small
```

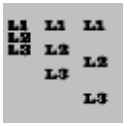
```
// Enlarging a font created at 12 pixels
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
textSize(32);
fill(0);
text("LNZ", 0, 40);   // Large
textSize(18);
text("STN", 0, 75);   // Medium
textSize(12);
text("BOS", 0, 100);  // Small
```

13-08

The `textLeading()` function sets the spacing between lines of text:

```
textLeading(dist)
```

The *`dist`* parameter defines this space in units of pixels.




```
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
String lines = "L1 L2 L3";
textLeading(10);
fill(0);
text(lines, 5, 15, 30, 100);
textLeading(20);
text(lines, 36, 15, 30, 100);
textLeading(30);
text(lines, 68, 15, 30, 100);
```

13-09

Letters and words can be drawn from their center, left, and right edges. The `textAlign()` function sets the alignment for drawing text:

```
textAlign(MODE)
```

The *`MODE`* parameter can be `LEFT`, `CENTER`, or `RIGHT`. It sets the display characteristics of the letters in relation to the value of the *`x`* parameter used in the `text()` function.

The settings for `textSize()`, `textLeading()`, and `textAlign()` will be used for all subsequent calls to the `text()` function. However, note that the `textSize()` function will reset the text leading, and the `textFont()` function will reset both the size and the leading.

13-10

```
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
line(50, 0, 50, 100);
fill(0);
textAlign(LEFT);
text("Left", 50, 20);
textAlign(RIGHT);
text("Right", 50, 40);
textAlign(CENTER);
text("Center", 50, 80);
```

The `textWidth()` function calculates and returns the pixel width of any character or text string. This number is calculated from the current font and size as defined by the `textFont()` and `textSize()` functions. Because the letters of every font are a different size and letters within many fonts have different widths, this function is the only way to know how wide a string or character is when displayed on screen. For this reason, always use `textWidth()` to position elements relative to text, rather than hard-coding them into your program.

13-11

```
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
char c = 'U';
float cw = textWidth(c);
text(c, 22, 40);
rect(22, 42, cw, 5);
String s = "UC";
float sw = textWidth(s);
text(s, 22, 76);
rect(22, 78, sw, 5);
```

Exercises

1. *Explore different typefaces in Processing. Draw your favorite word to the display window in your favorite typeface.*
2. *Draw a paragraph of text to the display window. Carefully select the composition.*
3. *Use two different typefaces to display the dialog between two characters.*

Notes

1. Ellen Lupton, *Thinking with Type: A Critical Guide for Designers, Writers, Editors, & Students* (Princeton Architectural Press, 2004), p. 29.

Math 3: Trigonometry

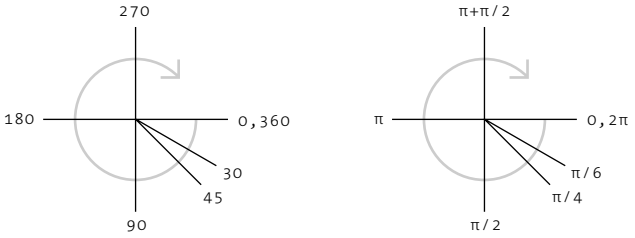
This unit introduces the basics of trigonometry and how to utilize it for generating form.

Syntax introduced:
`PI, QUARTER_PI, HALF_PI, TWO_PI, radians(), degrees()`
`sin(), cos(), arc()`

Trigonometry defines the relationships between the sides and angles of triangles. The trigonometric functions sine and cosine generate repeating numbers that can be used to draw waves, circles, arcs, and spirals.

Angles, Waves

Degrees are a common way to measure angles. A right angle is 90°, halfway around a circle is 180°, and the full circle is 360°. In working with trigonometry, angles are measured in units called radians. Using radians, the angle values are expressed in relation to the mathematical value π, written in Latin characters as "pi" and pronounced "pie." In terms of radians, a right angle is π/2, halfway around a circle is simply π, and the full circle is 2π.



Degree values Radian values

The numerical value of π is a constant thought to be be infinitely long and without a repeating pattern. It is the ratio of the circumference of a circle to its diameter. When writing Processing code, use the mathematical constant `PI` to represent this number. Other commonly used values of π are expressed with the constants `QUARTER_PI`, `HALF_PI`, and `TWO_PI`. Run the following line of code to see the value of π to 8 significant digits.

```
println(PI);  // Prints the value of PI to the text area
```
14-01

In casual use, the numerical value of π is 3.14, and 2π is 6.28. Angles can be converted from degrees to radians with the `radians()` function, or vice versa using `degrees()`.

This short program demonstrates the conversions between these representations:

14-02

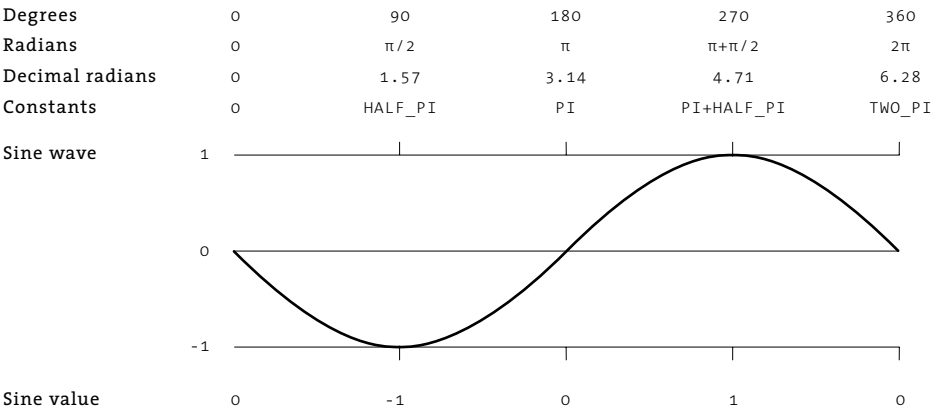
```
float r1 = radians(90);
float r1 = radians(180);
println(r1);  // Prints "1.5707964"
println(r2);  // Prints "3.1415927"
float d1 = degrees(PI);
float d2 = degrees(TWO_PI);
println(d1);  // Prints "180.0"
println(d2);  // Prints "360.0"
```

If you prefer working with degrees, use the `radians()` function in your programs to convert the degree values for use with functions that require radian values.

The `sin()` and `cos()` functions are used to determine the sine and cosine value of any angle. Each of these functions requires one parameter:

```
sin(angle)
cos(angle)
```

The *`angle`* parameter is always specified as a radian value. The values returned from these functions are always between the floating-point values of –1.0 and 1.0. The relationship between sine values and angles are shown here:



As angles increase in value, the sine values repeat. At the angle 0.0, the value of sine is also 0.0, and this value decreases as the angle increases. When the angle reaches 90.0° (π/2), the sine value increases until is it zero again at the angle 180.0° (π), then it continues to increase until the angle reaches 270.0° (π + π/2), at which point it begins decreasing until the angle reaches 360.0° (2π). At this point, the values repeat the cycle. The sine values can been seen by putting a `sin()` function inside a `for` structure and iterating while changing the angle value:

14-03

```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
  println(sin(angle));
}
```

Because the values from `sin()` are numbers between –1.0 and 1.0, they are easy to use in controlling a composition. Multiplying the numbers by 50.0, for example, will return values between –50.0 and 50.0.

14-04

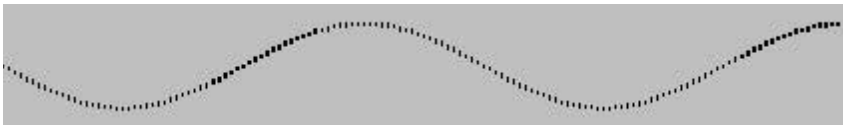
```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
  println(sin(angle) * 50.0);
}
```

To convert the sine values to a range of positive numbers, first add the value 1.0 to create numbers between 0.0 and 2.0. Divide that number by 2.0 to get a number between 0.0 and 1.0, which can then be simply remapped to any range. Alternatively, the `map()` function can be used to convert the values from `sin()` to any range. In this example, the values from `sin()` are put into the range between 0 and 1000.

14-05

```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
  float newValue = map(sin(angle), -1, 1, 0, 1000);
  println(newValue);
}
```

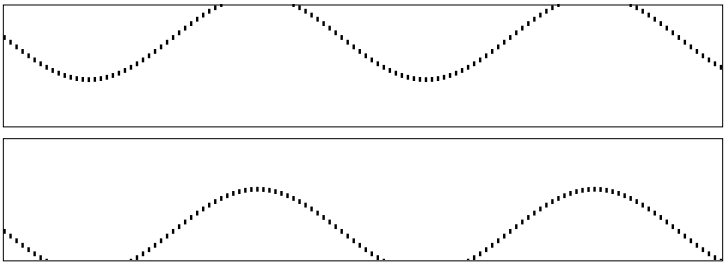
If we set the y-coordinates for a series of points with the numbers returned from the `sin()` function and continually increase the value of the angle parameter before each new coordinate is calculated, the sine wave emerges:

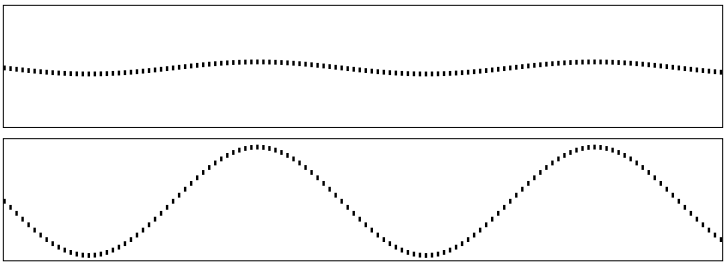


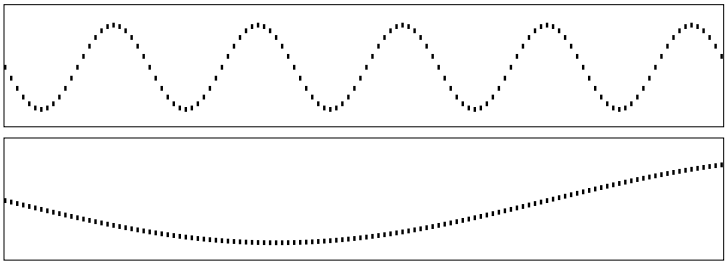
14-06

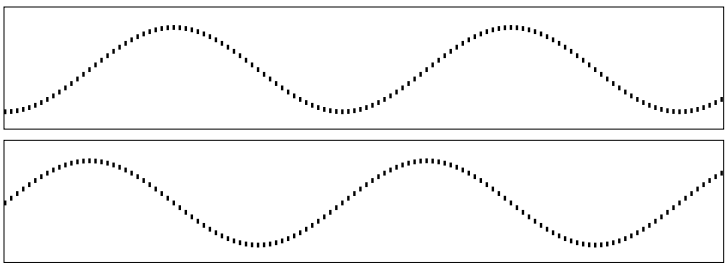
```
size(700, 100);
noStroke();
fill(0);
float angle = 0.0;
for (int x = 0; x <= width; x += 5) {
  float y = 50 + (sin(angle) * 35.0);
  rect(x, y, 2, 4);
  angle += PI/40.0;
}
```

Replacing some fixed numbers in the previous program with variables allows you to control the waveform by simply changing the values of the variables. The `offset` variable controls the y-coordinates of the wave, the `scaleVal` variable controls the









Modulating a sine wave
Different values for the variables in code 14-07 create a range of waves. Notice how each variable affects a different attribute of the wave.

height of the wave, and the `angleInc` variable controls the speed at which the angle increases, thereby creating a wave with a higher or lower frequency.

14-07

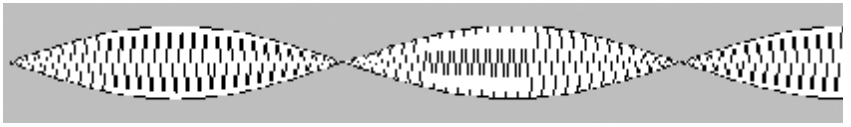
```
size(700, 100);
noStroke();
smooth();
fill(0);
float offset = 50.0;       // Y offset
float scaleVal = 35.0;     // Scale value for the wave magnitude
float angleInc = PI/28.0;  // Increment between the next angle
float angle = 0.0;         // Angle to receive sine values from
for (int x = 0; x <= width; x += 5) {
  float y = offset + (sin(angle) * scaleVal);
  rect(x, y, 2, 4);
  angle += angleInc;
}
```

The `cos()` function returns values in the same range and pattern as `sin()`, but the numbers are offset by π/2 radians (90°).

14-08

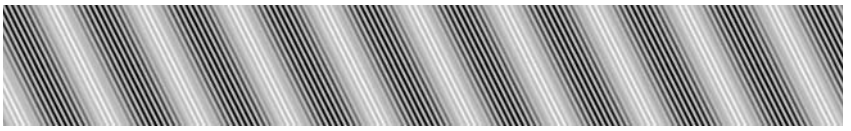
```
size(700, 100);
noStroke();
smooth();
float offset = 50.0;
float scaleVal = 20.0;
float angleInc = PI/18.0;
float angle = 0.0;
for (int x = 0; x <= width; x += 5) {
  float y = offset + (sin(angle) * scaleVal);
  fill(255);
  rect(x, y, 2, 4);
  y = offset + (cos(angle) * scaleVal);
  fill(0);
  rect(x, y, 2, 4);
  angle += angleInc;
}
```

The following examples demonstrate ways to use the numbers from the `sin()` function to generate shapes.



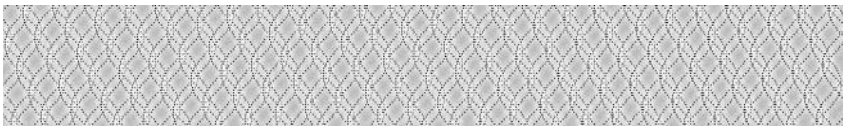
14-09

```
size(700, 100);
float offset = 50;
float scaleVal = 30.0;
float angleInc = PI/56.0;
float angle = 0.0;
beginShape(TRIANGLE_STRIP);
for (int x = 4 ; x <= width+5; x += 5) {
  float y = sin(angle) * scaleVal;
  if ((x % 2) == 0) {  // Every other time through the loop
    vertex(x, offset + y);
  } else {
    vertex(x, offset - y);
  }
  angle += angleInc;
}
endShape();
```



14-10

```
size(700, 100);
smooth();
strokeWeight(2);
float offset = 126.0;
float scaleVal = 126.0;
float angleInc = 0.42;
float angle = 0.0;
for (int x = -52; x <= width; x += 5) {
  float y = offset + (sin(angle) * scaleVal);
  stroke(y);
  line(x, 0, x+50, height);
  angle += angleInc;
}
```



14-11

```
size(700, 100);
smooth();
fill(255, 20);
float scaleVal = 18.0;
float angleInc = PI/28.0;
float angle = 0.0;
for (int offset = -10; offset < width+10; offset += 5) {
  for (int y = 0; y <= height; y += 2) {
    float x = offset + (sin(angle) * scaleVal);
    noStroke();
    ellipse(x, y, 10, 10);
    stroke(0);
    point(x, y);
    angle += angleInc;
  }
  angle += PI;
}
```

Circles, Arcs, Spirals

Circles can be drawn from sine and cosine waves. The example below has an angle that increments by 12°, all the way up to 360°. On each step, the `cos()` value of the angle is used to draw the x-coordinate, and the `sin()` value draws the y-coordinate. Because `sin()` and `cos()` return numbers between –1.0 and 1.0, the result is multiplied by the `radius` variable to draw a circle with radius 38. Adding 50 to the `x` and `y` positions sets the center of the circle at (50,50).

14-12

```
noStroke();
smooth();
int radius = 38;
for (int deg = 0; deg < 360; deg += 12) {
  float angle = radians(deg);
  float x = 50 + (cos(angle) * radius);
  float y = 50 + (sin(angle) * radius);
  ellipse(x, y, 6, 6);
}
```

If the angle is incremented only part of the way around the circle, an arc is drawn. For example, changing line 4 in the preceding program gives the following result:

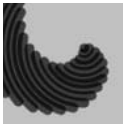
```
noStroke();                                                    14-13
smooth();
int radius = 38;
for (int deg = 0; deg < 220; deg += 12) {
  float angle = radians(deg);
  float x = 50 + (cos(angle) * radius);
  float y = 50 + (sin(angle) * radius);
  ellipse(x, y, 6, 6);
}
```

To simplify drawing arcs, Processing includes an `arc()` function:

```
arc(x, y, width, height, start, stop)
```

Arcs are drawn along the outer edge of an ellipse defined by the `x`, `y`, `width`, and `height` parameters. The `start` and `stop` parameters specify the angles needed to draw the arc form in units of radians. The following examples show the function in use.

```
strokeWeight(2);                                               14-14
arc(50, 55, 50, 50, 0, HALF_PI);
arc(50, 55, 60, 60, HALF_PI, PI);
arc(50, 55, 70, 70, PI, TWO_PI - HALF_PI);
noFill();
arc(50, 55, 80, 80, TWO_PI - HALF_PI, TWO_PI);
```



```
smooth();                                                      14-15
noFill();
randomSeed(0);
strokeWeight(10);
stroke(0, 150);
for (int i = 0; i < 160; i += 10) {
  float begin = radians(i);
  float end = begin + HALF_PI;
  arc(67, 37, i, i, begin, end);
}
```

To create a spiral, multiply the sine and cosine values by increasing or decreasing scalar values. In the following examples, the spiral grows as the `radius` variable increases:

14-16

```
noStroke();
smooth();
float radius = 1.0;
for (int deg = 0; deg < 360*6; deg += 11) {
  float angle = radians(deg);
  float x = 75 + (cos(angle) * radius);
  float y = 42 + (sin(angle) * radius);
  ellipse(x, y, 6, 6);
  radius = radius + 0.34;
}
```

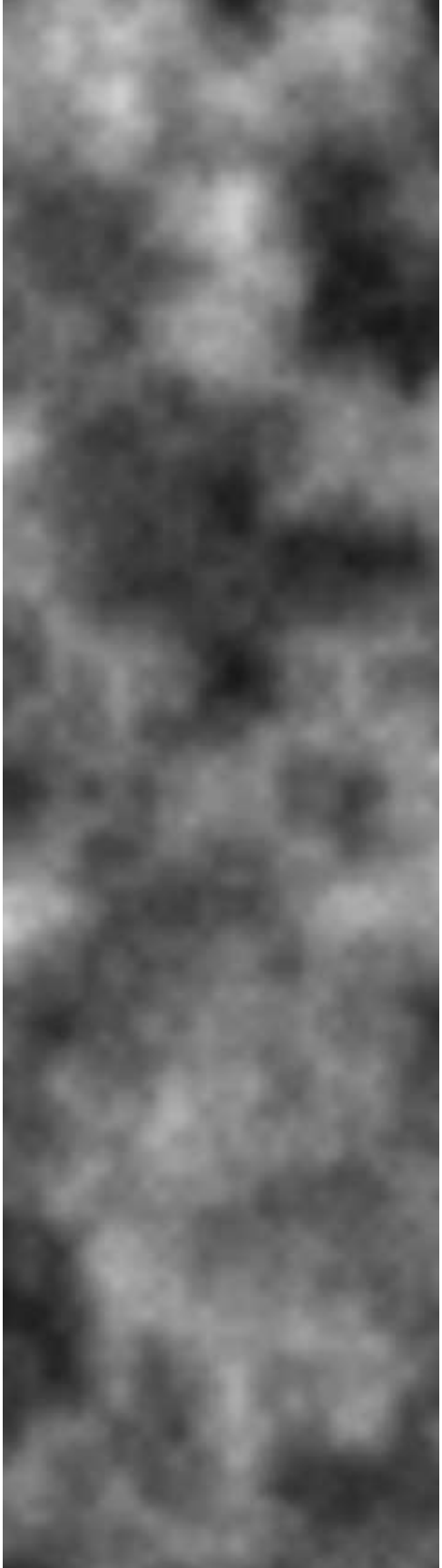
14-17

```
smooth();
float radius = 0.15;
float cx = 33;  // Center x- and y-coordinates
float cy = 66;
float px = cx;  // Start with center as the
float py = cy;  // previous coordinate
for (int deg = 0; deg < 360*5; deg += 12) {
  float angle = radians(deg);
  float x = cx + (cos(angle) * radius);
  float y = cy + (sin(angle) * radius);
  line(px, py, x, y);
  radius = radius * 1.05;
  px = x;
  py = y;
}
```

The content of this unit is applied to controlling movement in Motion 2 (p. 291).

Exercises

1. *Create a composition with the data generated using `sin()`.*
2. *Explore drawing circles and arcs with `sin()` and `cos()`. Develop a composition from the results of the exploration.*
3. *Generate a series of spirals and organize them into a composition.*



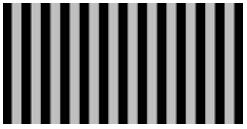
Math 4: Random

This unit introduces the basics of trigonometry and random numbers and explains how to utilize them for generating form.

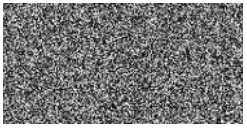
Syntax introduced:
`random(), randomSeed(), noise(), noiseSeed()`

Random compositional choices have a long history, particularly in modern art. In 1913 Marcel Duchamp's *3 Stoppages Étalon* employed the curves of dropped threads to derive novel units of measurement. Jean Arp used chance operations to define the position of elements in his collages. The composer John Cage sometimes tossed coins to determine the order and duration of notes in his scores. Artists integrate chance, randomness, and noise into their work either as a creative exercise or as a way of relinquishing some control to an external force. Actions like dropping, throwing, rolling, etc., deprive the artists of certain aspects of decisions. The world's chaos can be channeled into making images and objects with physical media. In contrast, computers are machines that make consistent and accurate calculations and must therefore simulate random numbers to approximate the kind of chance operations used in nondigital art.

There is an obvious contrast between rigid structure and complete chaos, and some of the most satisfying aesthetic experiences are created by infusing one with the other. The tension between order and chaos can actively engage our attention:



If a composition is obviously ordered, it will not hold attention beyond a quick glance.



Conversely, if a composition is entirely chaotic, it will also not retain one's gaze.

A balance between the two can yield a more satisfying result.

Unexpected values

The `random()` function is used to create unpredictable values within the range specified by its parameters.

```
random(high)
random(low, high)
```

When one parameter is passed to the function, it returns a `float` from zero up to (but not including) the value of the parameter. The function call `random(5.0)` returns

values from 0.0 up to 5.0. If two parameters are used, the function returns a value between the two parameters. Running `random(-5.0, 10.2)` returns values from -5.0 up to 10.2.

The numbers returned from `random()` are always floating-point values; therefore, they cannot be assigned to an `int` variable. The `int()` function can be used to convert a `float` value to an `int`.

15-01

```
float f = random(5.2);     // Assign f a float value from 0 to 5.2
int i = random(5.2);       // ERROR! Can't assign a float to an int
int j = int(random(5.2));  // Assign j an int value from 0 to 5
```

Because the numbers returned from `random()` are not predictable, each time the program is run, the result is different. The numbers from this function can be used to control almost any aspect of a program.

15-02

```
smooth();
strokeWeight(10);
stroke(0, 130);
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
```

The version of `random()` with two parameters provides more control over the results of the function. The previous example has been modified so the lines always progress from the upper-left to the lower-right, but the precise position is a chance operation. Storing the results of `random()` into a variable makes it possible to use the value more than once in the program. This program uses the random value `r` to set both the y-coordinate of the first point of the line and its stroke value.

15-03

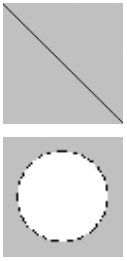
```
smooth();
strokeWeight(20);
stroke(0, 230);
float r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));
r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));
r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));
```

Using `random()` within a `for` structure is an easy way to generate random numbers.

15-04

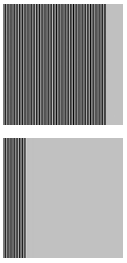
```
background(0);
stroke(255, 60);
for (int i = 0; i < 100; i++) {
  float r = random(10);
  strokeWeight(r);
  float offset = r * 5.0;
  line(i-20, 100, i+offset, 0);
}
```

To use random values to determine the flow of the program, you can place the `random()` function in a relational expression. In the first example below, either a line or an ellipse is drawn, depending on whether the result of the `random()` function is less than 50 or greater than or equal to 50, respectively. In the second example, between 1 and 50 vertical lines are drawn according to the result of the `random()` function.



15-05

```
float r = random(100);
if (r < 50.0) {
  line(0, 0, 100, 100);
} else {
  ellipse(50, 50, 75, 75);
}
```



15-06

```
int num = int(random(50)) + 1;
for (int i = 0; i < num; i++) {
  line(i * 2, 0, i * 2, 100);
}
```

It's sometimes desirable to include unpredictable numbers in your programs but to force the same sequence of numbers each time the program is run. The `randomSeed()` function is the key to producing such numbers.

randomSeed(value)

The *value* parameter must be an `int`. Use the same *value* parameter in a program each time it is run to force the same random numbers to be produced in the same order. There is a unique sequence of random values for every integer value. You might find that

using the number 1843 as the *value* parameter produces numbers that suit your needs, while the number 258 will not.

The following program is a slight variation on code 15-04. Adding `randomSeed()` ensures it will produce the same values every time it is run. Change the *value* parameter assigned to `randomSeed()` to generate a different set of numbers and thereby change the image produced by the program.

s=6

s=12

```
int s = 6;        // Seed value                                    15-07
background(0);
stroke(255, 60);
randomSeed(s);    // Produce the same numbers each time
for (int i = 0; i < 100; i++) {
  float r = random(10);
  strokeWeight(r);
  float offset = r * 5;
  line(i-20, 100, i+offset, 0);
}
```

Noise

The `noise()` function is a more controllable way to create unexpected values. It uses the Perlin Noise technique, developed by Ken Perlin.[1] Originally used for simulating natural textures through subtle irregularities, Perlin Noise is now also used for generating shapes and realistic motion. It works by interpolating between random values to create smoother transitions than the numbers returned from `random()`. The noise function has between one and three parameters:

```
noise(x)
noise(x, y)
noise(x, y, z)
```

The version of the function with one parameter is used to create a single sequence of random numbers. Additional parameters produce noise in more dimensions. For example, the version with two parameters can be used to create a two-dimensional texture. The version with three parameters can be used to create a three-dimensional shape or texture or an animated two-dimensional texture. Regardless of the number or value of the parameters, this function always returns values between 0.0 and 1.0. If other values are desired, an equation can be applied to the result to change the range (p. 81).

The numbers returned by noise can be made closer to or farther from the previous value by way of changes in the rate at which the parameter increases. As a general rule, the smaller the difference, the smoother the resulting noise sequence will be. A small change generates numbers that are closer to the previous value than a large increase would. Steps of 0.005–0.03 work best for most applications, but this will differ

depending on use. The following program uses the `inc` variable to define the difference between each number. Notice the differences between the results as the value of `inc` increases. The `noiseSeed()` function, which works like `randomSeed()`, is used to produce the same sequence of numbers each time the program runs.

inc=0.01

inc=0.1

15-08

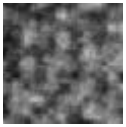
```
size(600, 100);
float v = 0.0;
float inc = 0.1;
noStroke();
fill(0);
noiseSeed(0);
for (int i = 0; i < width; i = i+4) {
  float n = noise(v) * 70.0;
  rect(i, 10 + n, 3, 20);
  v = v + inc;
}
```

Add a second parameter to `noise()` to open the possibility of creating a two-dimensional texture. In the following example, embedded `for` structures are used to generate continuous noise values on the x-axis and y-axis. The values returned from `noise()` are used to set the gray values for a grid of points drawn to the screen.

inc=0.04

inc=0.02

inc=0.1

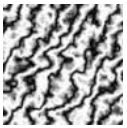


15-09

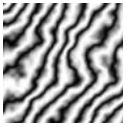
```
float xnoise = 0.0;
float ynoise = 0.0;
float inc = 0.04;
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    float gray = noise(xnoise, ynoise) * 255;
    stroke(gray);
    point(x, y);
    xnoise = xnoise + inc;
  }
  xnoise = 0;
  ynoise = ynoise + inc;
}
```

Diverse textures can be created using `noise()` in collaboration with `sin()`. The following example deforms a regular sequence of bars created with `sin()` into a texture reminiscent of one found in nature. The `power` variable sets the amount the texture deforms from the lines and the `density` parameter sets the granularity of the texture.

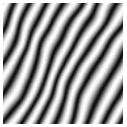
d=8



d=32



d=128



15-10

```
float power = 3;  // Turbulence power
float d = 8;      // Turbulence density
noStroke();
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    float total = 0.0;
    for (float i = d; i >= 1; i = i/2.0) {
      total += noise(x/d, y/d) * d;
    }
    float turbulence = 128.0 * total / d;
    float base = (x * 0.2) + (y * 0.12);
    float offset = base + (power * turbulence / 256.0);
    float gray = abs(sin(offset)) * 256.0;
    stroke(gray);
    point(x, y);
  }
}
```

Examples showing noise used for motion are given in Motion 2 (p. 291).

Exercises

1. *Use three variables assigned to random values to create a composition that is different every time the program is run.*
2. *Create a composition using a `for` structure and `random()` to make a composition of a different density every time the program is run.*
3. *Use `noise()` and `noiseSeed()` to create the same irregular shape every time a program is run.*

Notes

1. Perlin Noise was developed in the 1980s, and in 1997 Perlin received an Academy Award for Technical Achievement for this research. See *http://mrl.nyu.edu/~perlin/doc/oscar.html* and *http://www.noisemachine.com/talk1*.

Transform 1: Translate, Matrices

This unit introduces coordinate system transformations and explains how to control their scope.

Syntax introduced:
`translate(), pushMatrix(), popMatrix()`

The coordinate system introduced in Shape 1 uses the upper-left corner of the display window as the origin with the x-coordinates increasing to the right and the y-coordinates increasing downward. This system can be modified with transformations. The coordinates can be translated, rotated, and scaled so shapes are drawn to the display window with a different position, orientation, and size.

Translation

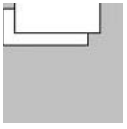
The `translate()` function moves the origin from the upper-left corner of the screen to another location. It has two parameters. The first is the x-coordinate offset and the second is the y-coordinate offset:

```
translate(x, y)
```

The values of the *x* and *y* parameters are added to any shapes drawn after the function is run. If 10 is used as the *x* parameter and 30 is used as the *y* parameter, a point drawn at coordinate (0,5) will instead be drawn at coordinate (10,35). Only elements drawn after the transformation are affected. The following examples show how this works.

16-01

```
// The same rectangle is drawn, but only the second is
// affected by translate() because it is drawn after
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts 10 pixels right and 30 down
rect(0, 5, 70, 30);
```



16-02

```
// A negative number used as a parameter to translate()
// moves the coordinates in the opposite direction
rect(0, 5, 70, 30);
translate(10, -10);  // Shifts 10 pixels right and up
rect(0, 5, 70, 30);
```

The `translate()` function is additive. If `translate(10,30)` is run twice, all the elements drawn after will display with an x-offset of 20 and a y-offset of 60.

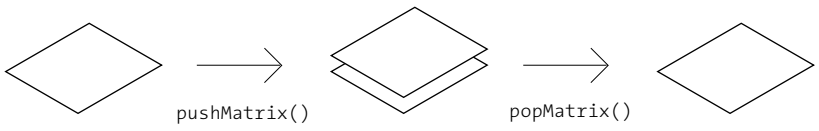
16-03

```
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts 10 pixels right and 30 down
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts everything again for a total
rect(0, 5, 70, 30); // 20 pixels right and 60 down
```

Controlling transformations

The transformation matrix is a set of numbers that defines how geometry is drawn to the screen. Transformation functions such as `translate()` alter the numbers in this matrix and cause the geometry to draw differently. In the previous examples, we saw how transformations accumulate as the program runs. The `pushMatrix()` function records the current state of all transformations so that a program can return to it later. To return to the previous state, use `popMatrix()`.

Think of each matrix as a sheet of paper with the current list of transformations (translate, rotate, scale) written on the surface. When a function such as `translate()` is run, it is added to the paper. To save the current matrix for later use, add a new sheet of paper to the top of the pile and copy the information from the sheet below. Any new changes are made to the top sheet of paper, preserving the numbers on the sheet(s) below. To return to a previous coordinate matrix, simply remove and discard the top sheet of paper to reveal the saved transformations below:



This is essentially how coordinate matrices are updated and stored, but more technical terms are used. Adding a sheet of paper is pushing, removing a sheet is popping and the pile of pages is called a stack. The `pushMatrix()` function is used to add a new coordinate matrix to the stack, and `popMatrix()` is used to remove one from the stack. Each `pushMatrix()` must have a corresponding `popMatrix()`. The function `pushMatrix()` cannot be used without `popMatrix()` and vice versa.

Compare the two examples below. Both draw the same rectangles, but with different results. The second example employs `pushMatrix()` and `popMatrix()` to isolate the effects of the `translate()` function to apply only to the first rectangle. Because the other rectangle is drawn after the call to `popMatrix()` it draws from its x-coordinate without being affected by the translation.

```
translate(33, 0);  // Shift 33 pixels right
rect(0, 20, 66, 30);
rect(0, 50, 66, 30);
```

16-04

```
pushMatrix();
translate(33, 0);  // Shift 33 pixels right
rect(0, 20, 66, 30);
popMatrix();  // Remove the shift
// This shape is not affected by translate() because
// the transformation is isolated between the pushMatrix()
// and popMatrix()
rect(0, 50, 66, 30);
```

16-05

Embedding the `pushMatrix()` and `popMatrix()` functions can further control their range. In the following example, the first rectangle is affected by the first translation, the second rectangle is affected by the first and second translations, and the third rectangle is only affected by the first translation because the second translation is isolated with a `pushMatrix()` and `popMatrix()` pair. The fourth rectangle is not affected by any of the translations because the `popMatrix()` on the second-to-last line cancels the `pushMatrix()` on the first line.

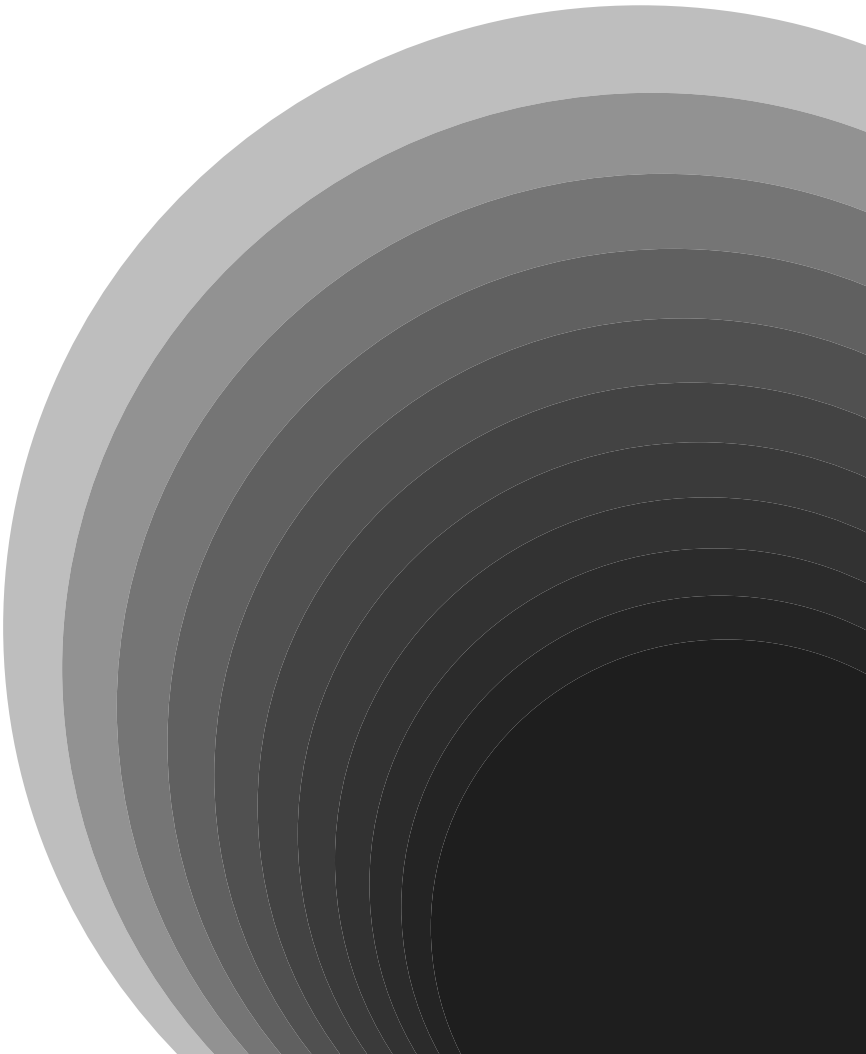
```
pushMatrix();
translate(20, 0);
rect(0, 10, 70, 20);  // Draws at (20, 30)
pushMatrix();
translate(30, 0);
rect(0, 30, 70, 20);  // Draws at (50, 30)
popMatrix();
rect(0, 50, 70, 20);  // Draws at (20, 50)
popMatrix();
rect(0, 70, 70, 20);  // Draws at (0, 70)
```

16-06

The transformation functions for rotating and scaling are introduced in Transform 2 (p. 137).

Exercises

1. *Use `translate()` to reposition a shape.*
2. *Use multiple translations to reposition a series of shapes.*
3. *Use `pushMatrix()` and `popMatrix()` to rearrange the composition from exercise 2.*



Transform 2: Rotate, Scale

This unit introduces the transformation functions for rotating and scaling and explains how to combine the functions to control the effect.

Syntax introduced:
`rotate(), scale()`

The transformation functions are powerful ways to modify the geometry displayed to the screen. It's simple to use one, but combining them requires a greater understanding of how they work. The order in which transformation functions are run can radically change the way they affect the coordinates.

Rotation, Scaling

The `rotate()` function rotates the coordinate system so that shapes can be drawn to the screen at an angle. It has one parameter that sets the amount of the rotation as an angle:

```
rotate(angle)
```

The rotate function assumes that the *angle* is specified in units of radians (p. 117). Shapes are always rotated around their position relative to the origin (0,0), and positive numbers rotate them in a clockwise direction.

As with all transformations, the effects of rotation are cumulative. If there is a rotation of π/4 radians and another of π/4 radians, objects drawn afterward will be rotated π/2 radians. The following examples show the most basic use of the `rotate()` function.

```
smooth();
rect(55, 0, 30, 45);
rotate(PI/8);
rect(55, 0, 30, 45);
```

17-01

```
smooth();
rect(10, 60, 70, 20);
rotate(-PI/16);
rect(10, 60, 70, 20);
rotate(-PI/8);
rect(10, 60, 70, 20);
```

17-02

These examples make it clear that rotating objects around the origin has limitations. To rotate an object at a different position, it's necessary to use `translate()` followed by `rotate()`. This is explained in the next section, "Combining transformations."

The `scale()` function magnifies the coordinate system so that shapes are drawn larger. It has one or two parameters to set the amount of increase or decrease:

```
scale(size)
scale(xsize, ysize)
```

The version with one parameter scales shapes in all dimensions, and the version with two parameters can scale the x-dimension separately from the y-dimension. The parameters to scale are defined in terms of percentages expressed as decimals. Examples of decimal percentages are 2.0 for 200%, 1.5 for 150%, and 0.5 for 50%. The following examples show the most basic use of the `scale()` function.

17-03

```
smooth();
ellipse(32, 32, 30, 30);
scale(1.8);
ellipse(32, 32, 30, 30);
```

17-04

```
smooth();
ellipse(32, 32, 30, 30);
scale(2.8, 1.8);
ellipse(32, 32, 30, 30);
```

As the previous examples show, the stroke weight is also affected by `scale()`. To keep the same stroke weight and scale a shape, divide the parameter of the `strokeWeight()` function by the scale value.

17-05

```
float s = 1.8;
smooth();
ellipse(32, 32, 30, 30);
scale(s);
strokeWeight(1.0 / s);
ellipse(32, 32, 30, 30);
```

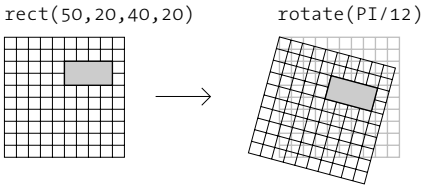
As with `translate()` and `rotate()`, the effects of each `scale()` accumulate each time the function is run.

17-06

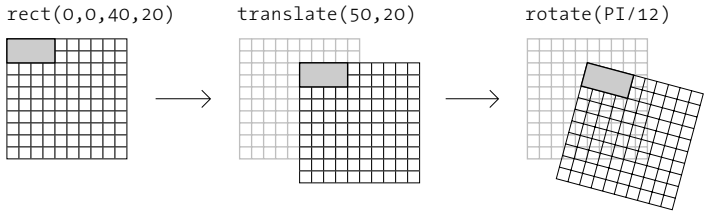
```
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
```

Combining transformations

When shapes are drawn to the screen, the `transform()`, `rotate()`, and `scale()` functions affect them in relation to the origin. For example, rotating a rectangle at coordinate (50,20) will cause the shape to orbit around the origin and not around its center or corner as you might expect:

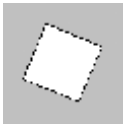


To rotate this shape around its upper-left corner, you must place that point at the coordinate (0,0). A translation is used to put the shape into the desired position in relation to the global coordinates. When the rotate function is run, the shape now orbits around its upper-left corner, the origin of its local coordinate system:



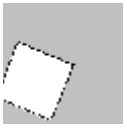
There are two ways to think about transformations. One method is to view the coordinate system as modified and the coordinates for shapes as converted to the new coordinate system. For example, if the coordinate system is rotated 30°, the coordinates of any shape drawn to the screen are converted into this modified system and displayed with a 30° tilt. The other school of thought applies the transformations directly to the shapes. In this same example, the shape itself is perceived to be rotated 30°.

The order in which transformations are made affects the results. The following two examples have the same lines of code, but the order of the `translate()` and `rotate()` functions is reversed :



```
translate(width/2, height/2);
rotate(PI/8);
rect(-25, -25, 50, 50);
```

17-07



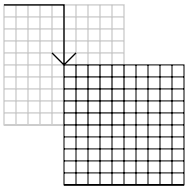
```
rotate(PI/8);
translate(width/2, height/2);
rect(-25, -25, 50, 50);
```

17-08

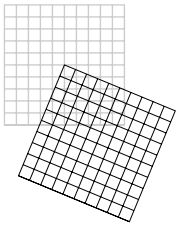
Code 17-07 analyzed from two perspectives

Coordinate view
Reading the code from top to bottom

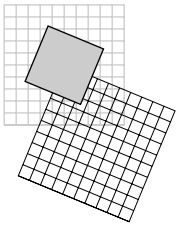
Translate



Rotate

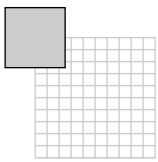


Draw rectangle

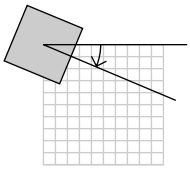


Shape view
Reading the code from bottom to top

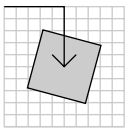
Draw rectangle



Rotate



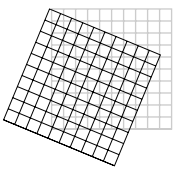
Translate



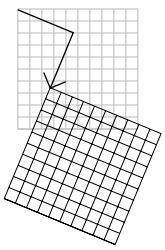
Code 17-08 analyzed from two perspectives

Coordinate view
Reading the code from top to bottom

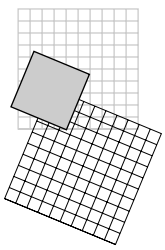
Rotate



Translate

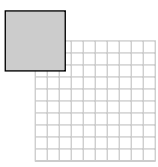


Draw rectangle

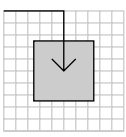


Shape view
Reading the code from bottom to top

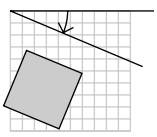
Draw rectangle



Translate



Rotate



Transformation combinations
The order in which transformations occur in a program affects how they combine. For example, a `rotate()` *after a* `translate()` *will have a different effect than the reverse. These diagrams present two ways to think about the transformations in codes 17-06 and 17-07.*

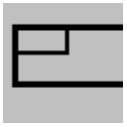
These simple examples demonstrate the potential in combining transformations but also make clear that transformations require thought and planning. More combined examples follow:

17-09

```
translate(10, 60);
rect(0, 0, 70, 20);
rotate(-PI/12);
rect(0, 0, 70, 20);
rotate(-PI/6);
rect(0, 0, 70, 20);
```

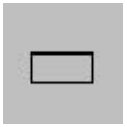
17-10

```
translate(45, 60);
rect(-35, -5, 70, 10);
rotate(-PI/8);
rect(-35, -5, 70, 10);
rotate(-PI/8);
rect(-35, -5, 70, 10);
```



17-11

```
noFill();
translate(10, 20);
rect(0, 0, 20, 10);
scale(2.2);
rect(0, 0, 20, 10);
scale(2.2);
rect(0, 0, 20, 10);
```



17-12

```
noFill();
translate(50, 30);
rect(-10, 5, 20, 10);
scale(2.5);
rect(-10, 5, 20, 10);
```

The effects of the transformation functions accumulate throughout the program, and these effects can be magnified with a `for` structure.

17-13

```
background(0);
smooth();
stroke(255, 120);
translate(66, 33);              // Set initial offset
for (int i = 0; i < 18; i++) {  // 18 repetitions
  strokeWeight(i);              // Increase stroke weight
  rotate(PI/12);                // Accumulate the rotation
  line(0, 0, 55, 0);
}
```

17-14

```
background(0);
smooth();
noStroke();
fill(255, 48);
translate(33, 66);              // Set initial offset
for (int i = 0; i < 12; i++) {  // 12 repetitions
  scale(1.2);                   // Accumulate the scaling
  ellipse(4, 2, 20, 20);
}
```

Working with these examples will be more helpful than reading the explanation over and over. Try these examples inside Processing and make modifications to the numbers used and the sequence of translate, rotate, and scale to develop a sense of how these functions work.

New coordinates

The default position of the coordinate origin (0,0) is the upper-left corner of the display window, the x-coordinate numbers increase to the right, the y-coordinates increase from the top, and each coordinate maps directly to a pixel position. The transformation functions can change these defaults to modify the coordinate system. The following examples move the origin to the center and lower-left corner of the display window and modify the scale.

17-15

```
// Shift the origin (0,0) to the center
size(100, 100);
translate(width/2, height/2);
line(-width/2, 0, width/2, 0);    // Draw x-axis
line(0, -height/2, 0, height/2);  // Draw y-axis
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 45, 45);  // Draw at the origin
ellipse(-width/2, height/2, 45, 45);
ellipse(width/2, -height/2, 45, 45);
```

The `translate()` and `scale()` functions can combine to change the range of values. In the following example, the right edge of the screen is mapped to the x-coordinate of 1.0, the left edge to the x-coordinate –1.0, the top edge to the y-coordinate 1.0, and the bottom edge to the y-coordinate –1.0. This system will always scale to fit the entire display window. Run this program, but change the parameters to `size()` to see it work.

17-16

```
// Shift the origin (0,0) to the center
// and resizes the coordinate system
size(100, 100);
scale(width/2, height/2);
translate(1.0, 1.0);
strokeWeight(1.0/width);
line(-1, 0, 1, 0);  // Draw x-axis
line(0, -1, 0, 1);  // Draw y-axis
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 0.9, 0.9);  // Draw at the origin
ellipse(-1.0, 1.0, 0.9, 0.9);
ellipse(1.0, -1.0, 0.9, 0.9);
```

The `translate()` and `scale()` functions can be combined to put the origin in the lower-left corner of the screen. This is the coordinate system used by Adobe Illustrator and PostScript. Scaling the y-axis by –1 causes the y-coordinates to increment in the opposite direction. This can be useful when converting a program written using this coordinate system into Processing, rather than converting the y-coordinate of every point.

17-17

```
// Shift the origin (0,0) to the lower-left corner
size(100, 100);
translate(0, height);
scale(1.0, -1.0);
line(0, 1, width, 1);    // Draw x-axis
line(0, 1, 0, height );  // Draw y-axis
smooth();
noStroke();
fill(255, 204);
ellipse(0, 0, 45, 45);   // Draw at the origin
ellipse(width/2, height/2, 45, 45);
ellipse(width, height, 45, 45);
```

Exercises

1. *Use `rotate()` to change the orientation of a shape.*
2. *Use `scale()` with a `for` structure to scale a shape multiple times.*
3. *Combine `translate()` and `rotate()` to rotate a shape around its own center.*

Development 1: Sketching, Techniques

This unit discusses the idea of sketching code and the iterative development process.

There are similarities between learning a programming language and learning a new spoken language. Initially, one learns basic elements of a spoken language—such as simple words and grammar rules—and mimics short phrases. Learning to communicate ideas and express emotions within the language takes more time. Similarly, the first step in computer programming is to understand the basic elements such as comments, variables, and functions. The next step is to learn to read and modify simple example programs. Later, one begins to write programs from scratch. The most interesting and difficult stage of learning to program comes later, as one gains the ability to put the language elements together to express ideas about form, motion, and behavior. Like learning a foreign language, becoming fluent in a programming language can take years.

Sketching software

Sketching ranges from informal exploration to focused refinement. It is used to create many variations within a short period of time, or to develop a specific idea. Sketching forces the definition of vague ideas by making them physical. Sketches are powerful communication tools—they can get ideas out of one's head and into a format that can be better understood by others.

It is important to work out ideas on paper before investing time in writing code. Paper and pencil allow for fast iteration in the early stages of a project. The most important aspect of programming is figuring out what will be created and how it will function, so working out these ideas away from a computer keeps the focus on an idea, rather than on its implementation.

A good paper sketch for software will include a series of images that demonstrate how the narrative structure of the piece works, much like an animator's storyboard. In addition to images that will appear on screen, sketches often contain diagrams of the program's flow, data elements, and notation for showing how forms will move and interact. Programs can also be planned using combinations of image mock-ups, formal schematics, and text descriptions.

After refined ideas reach a point where working on paper is no longer useful, code can continue the development. The first step in creating code is a continuation of the sketching process. Write short pieces of code independently before worrying about the structure of the larger program. Writing small, focused programs makes a developer better at writing code when it matters most: when working on a more refined implementation.

Processing programs are called sketches to emphasize this method of working. The Processing sketchbook is a way of storing and organizing programs. Code sketches can be reviewed and developed incrementally like drawings in a paper sketchbook. Ideas that flash by while walking or just after waking up can be quickly made into code and stored for future use. The Processing environment encourages this type of writing because one need only press the New button in the toolbar to start a new sketch.

Some programming languages encourage a sketching approach, and others make it difficult. Scripting languages, such as Perl or Python, are designed to encourage rapid development at the expense of running speed and control. Processing is not a scripting language, but is designed to "feel" like a scripting language while providing the same capabilities as a more complete language like Java. This topic is discussed in more depth in Appendix F (p. 679).

Programming techniques

There are as many ways to write programs as there are people who write software. Some common strategies for creating programs include modification, augmentation, collage, and writing code from scratch. People learning to code often expect most programs to be written from scratch, but that's rarely the case, particularly for the style of work built with Processing. Learning to read and modify code helps programmers increase their skills. Even advanced programmers work from others' examples when learning new techniques.

Modification
Changing the values of variables in existing programs is a good way to explore code. Programs can be modified by trial and error or more deliberately. One way to start understanding a program is to change slightly the value of one variable and then run the code to see the result. If there is no obvious difference, change the value again. Making the correlation between a variable and a change in the way a program runs is a good first step to understanding how it works. Disabling lines of code by placing them inside a `/*` and `*/` comment block (called "commenting out") is another way to decrypt a program. A little understanding of the way a program is structured can facilitate logical guesses about what different lines of code are doing. These modification techniques aid in learning new skills or parsing an example. Making small changes to an existing program encourages exploration and getting a feel for the code.

Augmentation
Augmentation uses existing code as a base for further exploration. It is similar to but more ambitious than modification. Generic program examples can serve as a foundation for longer, more specific programs. An example that draws a Bézier curve can be used as a base for drawing a series of curves (as shown in Shape 2, p. 69). A program that displays a photograph can form the basis of a photo montage application. Sample programs

provide a concise reminder of syntax. The spartan programs presented in this book provide a broad base for making enhancements.

Collage
The collage technique involves cutting and pasting elements of different programs together to create a new program. It's analogous to creating music by sampling or making a visual collage from newspaper and magazine clippings. In order to avoid errors, combine code carefully by copying a few lines of code at a time and running the program to make sure it's always working. Copying large portions of code can introduce a number of simultaneous errors. Mindlessly copying and pasting code can create "Frankenstein" code that's difficult to debug. As an individual's knowledge of programming increases, using this technique becomes easier, and common problems can be avoided, such as adding multiple copies of the same method, like `draw()` or `setup()`, to the code.

Coding from scratch
Rarely do programmers write a complete program entirely from scratch. At the minimum, most people start with a template. A template is an outline with code infrastructure common to many programs. Sometimes it's not possible to find a related example or appropriate template and it's necessary to start with a blank page. In this case, comments are a great way to start building a program. Comments can be used to build an outline of the program's intention, logic, and flow. After this structure has been defined, lines of code can be slowly added and run in an attempt to realize those decisions.

Regardless of the technique used for programming, writing a few lines of code or making only a few changes at a time is a good tactic. Entering many lines of untested code before running the program increases the potential for multiple errors. The more errors in a program, the more difficult they become to find. Running a growing program piece by piece reduces the chance for multiple errors. As your comfort with code and your skills increase, it becomes possible to make more modifications between tests.

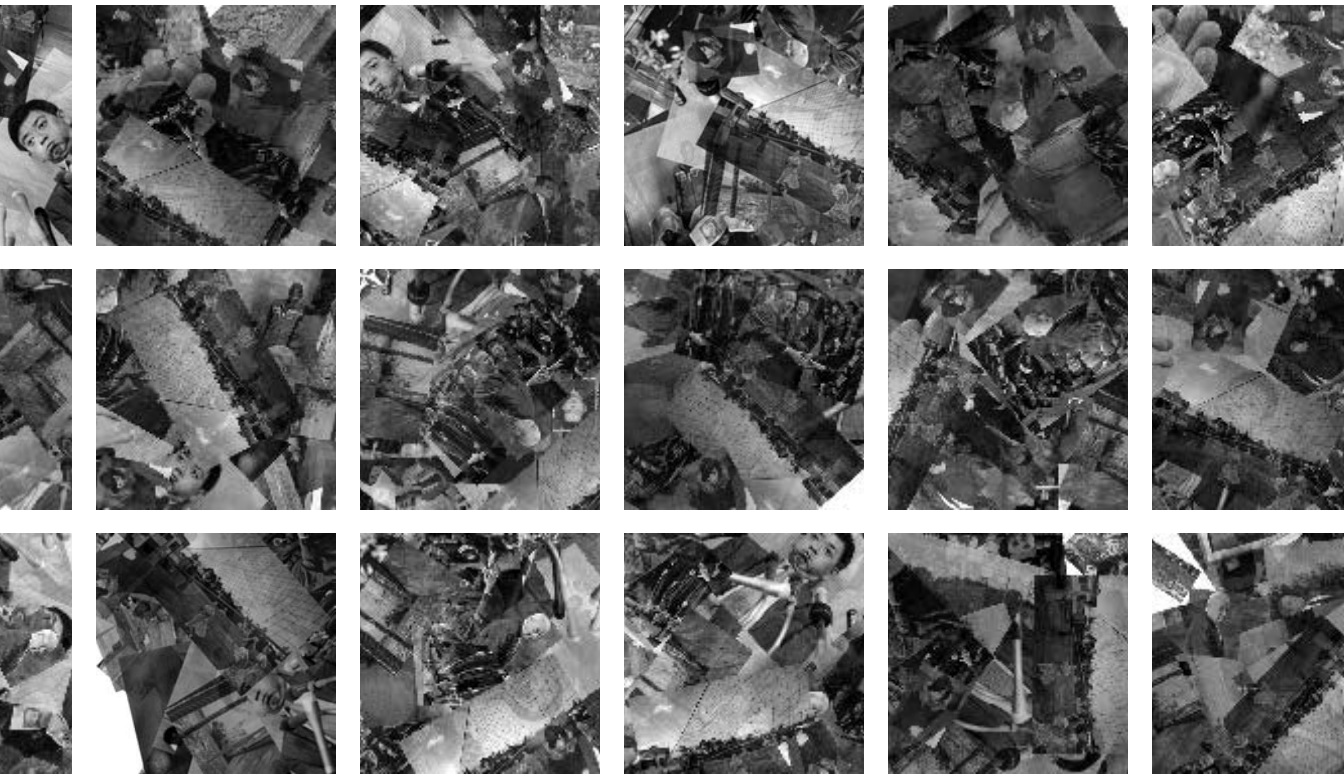
Synthesis 1: Form and Code

This unit presents examples of synthesizing concepts from Structure 1 though Transform 2.

The previous units introduced concepts and techniques including coordinates, drawing with vertices, variables, iteration, conditionals, trigonometry, and transformations. Understanding each of these in isolation is the first step toward learning how to program. Learning how to combine these elements is the second step. There are many ways to combine the components of every programming language for purposes of communication and expression. This programming skill is best acquired through writing more ambitious software and reading more complex programs written by others. This unit introduces four new programs that push beyond those on the previous pages.

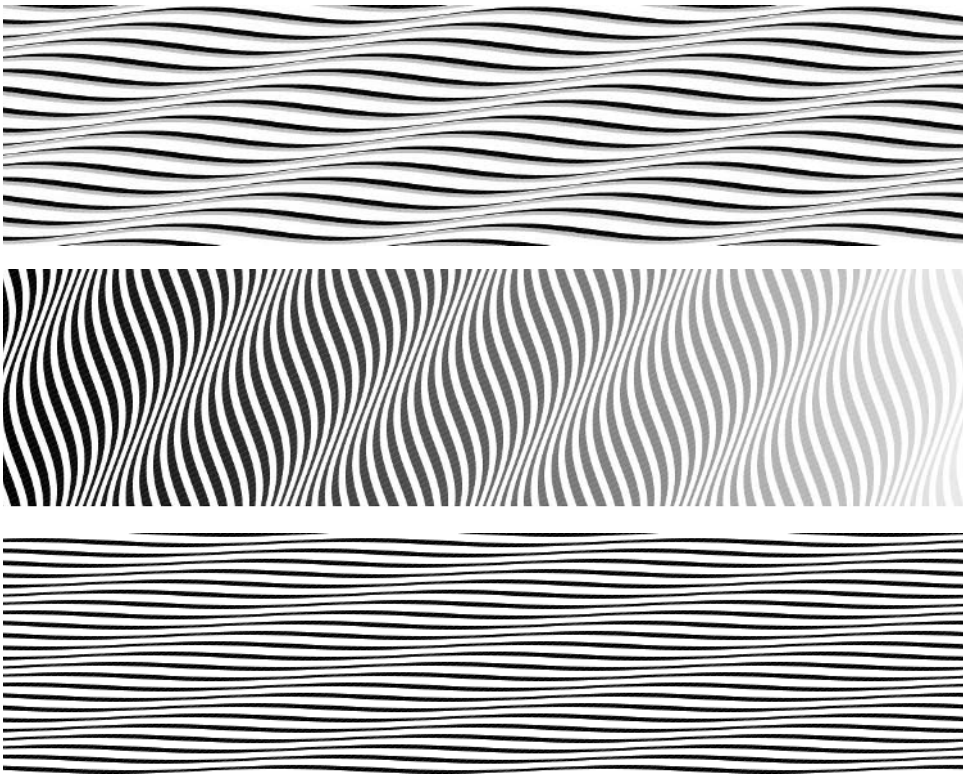
Artists and designers throughout the twentieth century practiced the ideas and visual styles currently associated with software culture, long before personal computers became a common tool. The aesthetic legacies of the Bauhaus, art deco, modernist architecture, and op art movements retain a strong voice in contemporary culture, while new forms have emerged through software explorations within the scientific and artistic communities. The programs in this unit reference images from the last hundred years; sampling from Dadaist collage, optical paintings, a twenty-year-old software program, and mathematics.

The software featured in this unit is longer than the brief examples given in this book. It's not practical to print it on these pages, but the code is included in the Processing code download at www.processing.org/learning.

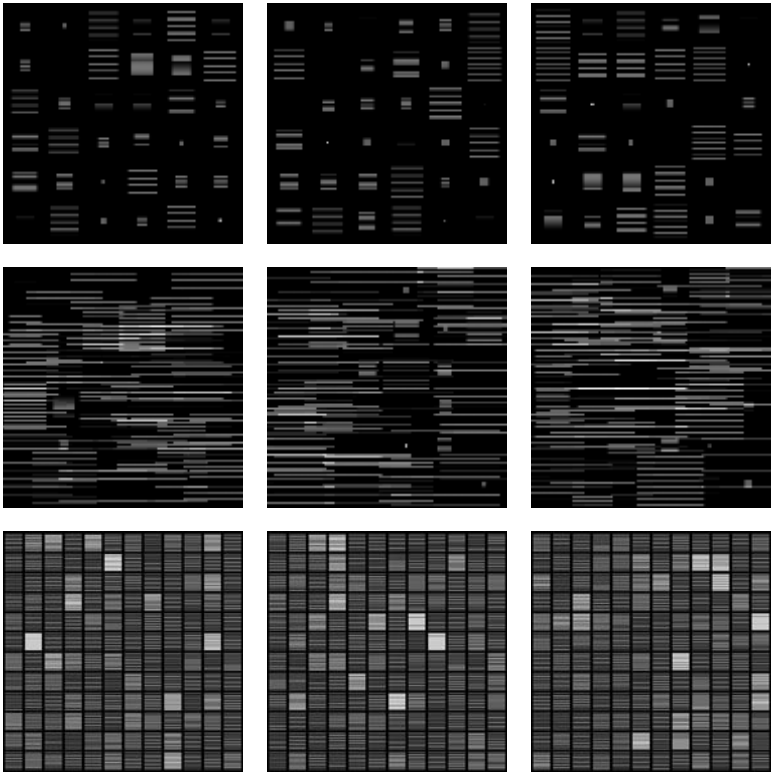


Collage Engine. Reacting to the horror of World War I, European artists and poets within the Dada cultural movement produced works that were deliberately irrational and absurd and that rejected the current standards of art. The poet Tristan Tzara devised a technique for writing that involved taking text from the newspaper, separating the individual words, and putting them back together in random order.

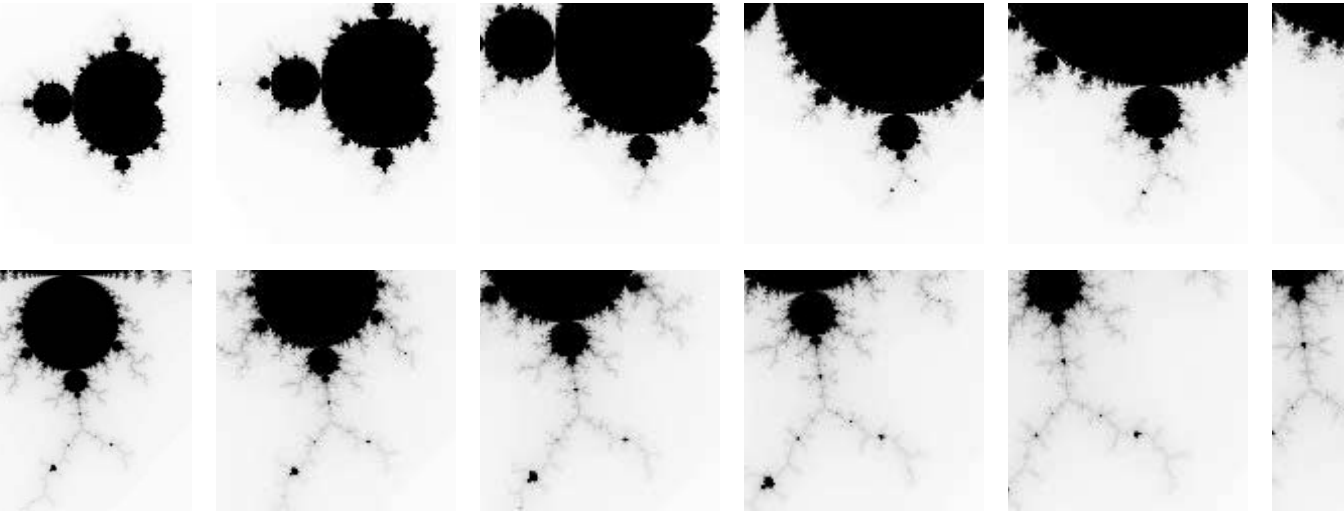
The images shown here were produced using a similar technique with photographs from the first section of *The New York Times* of 9 June 2006. The pictures were cut, scanned, and then repositioned randomly to produce these collages.



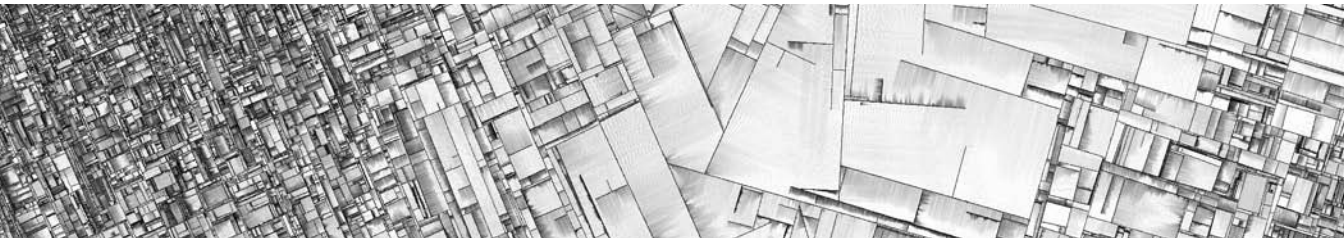
Riley Waves. These images were influenced by the paintings of Bridget Riley, a British artist who has exhibited her work since the mid-1960s. Riley's optically vibrant works often have a strong emotional and visceral effect on the viewer. She works exclusively with simple geometric shapes such as curves and lines and constructs visual vibrations through repetition. Because each of the waves in these images transitions from thick to thin, only the `beginShape()` and `endShape()` functions could create them. Like code 14-09 (p. 122), each wave is comprised of a sequence of triangles drawn using the `TRIANGLE_STRIP` parameter.



Wilson Grids. In his 1985 book *Drawing with Computers*, the artist Mark Wilson surveys the technology of that era and presents many examples of code for drawing to pen plotters and screens. These images were created from code converted from Wilson's programs (written in the BASIC language) to Processing. They utilize the embedded `for` technique introduced in code 6-07 (p. 65).



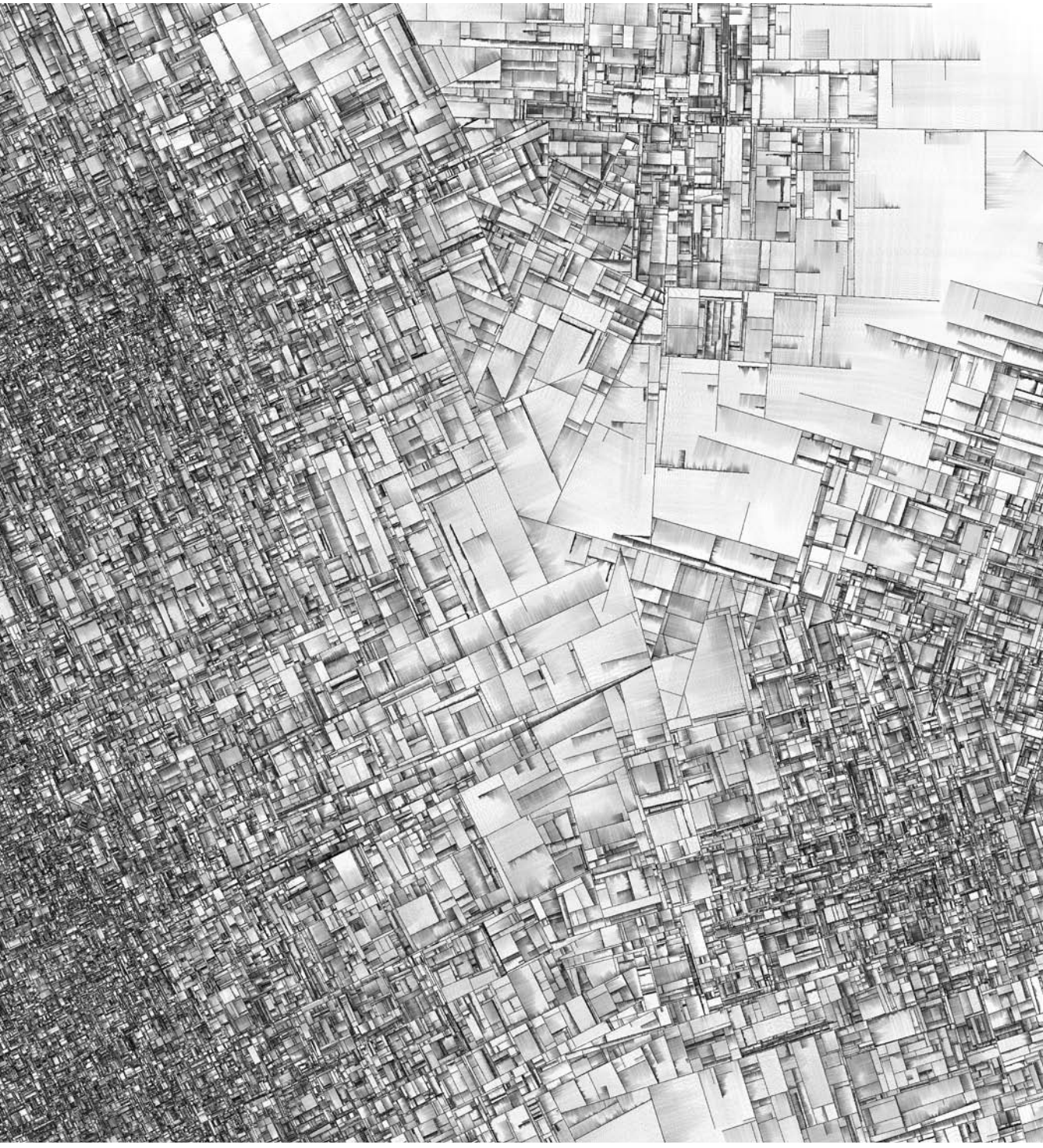
Mandelbrot Set. A fractal is a shape that appears similar at different scales. Examples of fractals in nature include clouds, mountains, and the network of blood vessels. *Fractal* is a term coined by the mathematician Benoit Mandelbrot, who also devised the Mandelbrot set, an equation that defines a fractal image. The Mandelbrot set left the confines of the mathematics community and entered into popular culture in the 1980s through the popularization of fractal images in books and magazines. These images of the Mandelbrot set were created by changing the scale to render the equation as pixels. The gray value for each pixel of the display window is determined through the equation.



Detail of *Substrate*, 2004. Image courtesy of Jared Tarbell.

Interviews 1: Print

Jared Tarbell. *Fractal.Invaders*, *Substrate*
Martin Wattenberg. *Shape of Song*
James Paterson. *The Objectivity Engine*
LettError. RandomFont Beowolf



Fractal.Invaders, Substrate *(Interview with Jared Tarbell)*

Creator	Jared Tarbell
Year	2004
Medium	Software, Prints
Software	Flash, Processing
URL	www.complexification.net

What are *Fractal.Invaders* and *Substrate*?

Fractal.Invaders *and* Substrate *are unique programs that both generate space-filling patterns on a two-dimensional surface. Each uses simplified algorithmic processes to render a more complex whole.*

Fractal.Invaders begins with a rectangular region and recursively fills it with little "invader" objects. Each invader is a combination of black squares arranged in a 5 × 5 grid generated at random during runtime. The only rule of construction requires that the left side of the invader be a mirror copy of the right side. This keeps them laterally symmetric, which endows them with a special attractiveness to the human eye.

There are a total of 32,768 (2^{15}) possible invaders. The magnitude of 15 comes from the product of 3 columns and 5 rows (the last 2 columns of the grid are ignored since they are the same as the first 2). The 2 comes from the fact that each space in the grid can be either black or white.

A small bit of interactivity allows each invader to be clicked. Clicking an invader destroys it, although the empty space left behind is quickly filled with smaller invaders. In this way, the user is ultimately doomed.

Substrate begins similarly with an empty rectangular region. It has been compared to crystal formation and the emergent patterns of urban landscapes. A single line (known internally as a "crack" since the algorithm was inspired by sunbaked mud cracks) begins drawing itself from some random point in some random direction. The line continues to draw itself until it either (a) hits the edge of the screen or (b) hits another line, at which point it stops and two more lines begin. The one simple rule used in the creation of new lines is that they begin at tangents to existing lines. This process is repeated until there are too many lines to keep track of or the program is stopped.

Before writing the program, I only had a vague idea of what it might look like. It wasn't until the first couple of bug-free executions that I realized something incredible was happening. The resulting form was much more complex than the originating algorithm. This particular quality of software is what keeps me interested.

Interesting effects can be created by introducing small variations in the way the first couple of lines are drawn. One of my favorite initial conditions is the creation of three lines, each in its own localized space with a direction that varies from the others by about 30 degrees. After growing for a short time into coherent lattices, they eventually crash into each other, creating an affluence of odd shapes and unexpected mazes.

The watercolor quality of the rendering is achieved by placing large numbers of mostly transparent pixels perpendicular to each line's growth. The trick is to deposit precisely the same

Substrate, 2004. Image courtesy of the artist.

number of pixels regardless of the length of the area being filled. This produces an interesting density modulation across an even mass of pixels.

Why did you create this software?

For me, one of the most enjoyable subjects in computer science is combination. I ask myself a question like, "Given some rules and a few simple objects, how many possible ways can they be combined?" Seldom can I answer this using thought alone, mainly because the complexity of even just a few elements is outside the realm of my imagination. Instead, I write computer programs to solve it for me. Fractal.Invaders *is definitely one of these questions, and is answered completely with the rendering of every single invader. Substrate asks a similar question but with results that, although beautiful, are a little less complete.*

What software tools were used?

For Fractal.Invaders, *I used a combination of Flash and custom software to create and capture the invaders, respectively. In Flash, all work was done using ActionScript. A single symbolic element (black square) exists in the library. Code takes this square and duplicates it hundreds of thousands of times. The entire generative process takes about five minutes to complete, depending on the size of the region to be filled and the speed of the execution. Capturing a high-resolution image of the result is accomplished with a program that scales the Shockwave Flash (SWF) file very large and saves the screen image out to a file.*

Substrate *was created entirely in Processing. Processing was particularly well suited for this as it excels at drawing, especially when dropping millions of deep-color pixels. Processing can also save out extremely large graphic images in an automated fashion. Oftentimes I will run a Processing project overnight. In the morning I awake to a vast collection of unique images, the best of which are archived as print editions.*

Why did you use these tools?

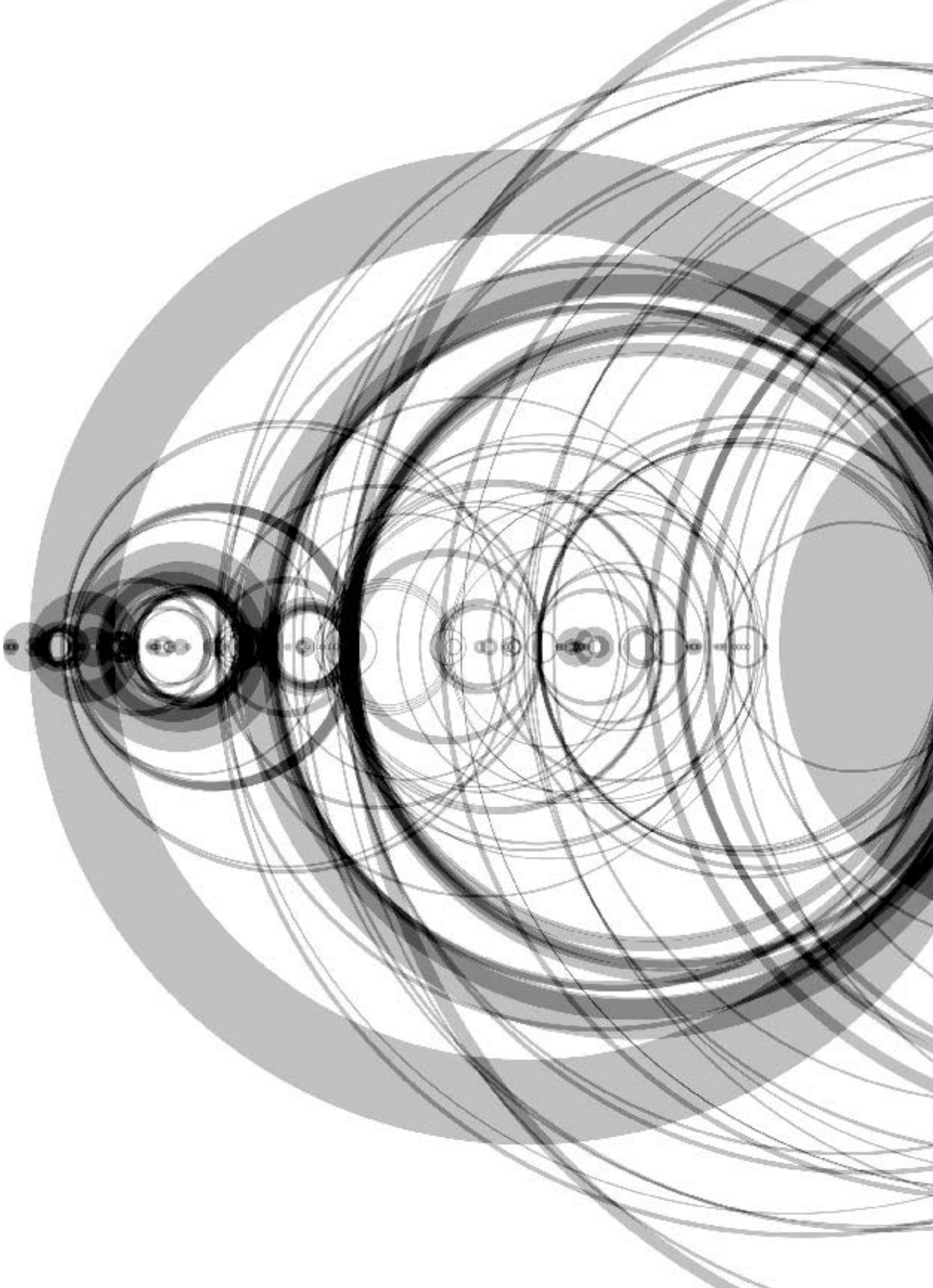
I use Flash because I am comfortable working within it. I use Processing because it enables me to do things Flash simply cannot. Both environments allow me to take a program from concept to completion in a number of hours. Complex visual logic can be built up without the bulky overhead required in more traditional graphic programming languages.

Flash excels at rendering very high resolution images nicely, displaying native vector objects with a high degree of precision and full antialiasing. Processing gives me the computational speed to increase the number of objects in the system by a magnitude of 20 or more. Both programs allow me to produce work that is capable of being viewed by a large number of people worldwide.

Why do you choose to work with software?

With software, anything that can be imagined can be built. Software has a mysterious, undefined border. Programming is truly a process of creating something from nothing. I enjoy most John Maeda's perspective: "While engaged in the deepest trance of coding, all one needs to wish for is any kind of numerical or symbolic resource, and in a flash of lightning it is suddenly there, at your disposal."

Fractal.Invaders, 2004. Image courtesy of the artist.



Shape of Song *(Interview with Martin Wattenberg)*

Creator	Martin Wattenberg
Year	2002
Medium	Software, Prints
Software	Java
URL	www.turbulence.org/Works/song

What is *Shape of Song*?

The Shape of Song *is an attempt to answer the seemingly paradoxical question "What does music look like?" The custom software in this work draws musical patterns in the form of translucent arches, allowing viewers to literally see the shape of a composition.*

One of the satisfying aspects of the visualization, to me, is that different musical styles translate to characteristic, distinct visual styles. Folk songs yield simple, repetitive arrangements. Classical pieces have an almost mathematically precise visual structure. Jazz translates to my favorite diagrams, for they often start out with extremely regular patterns and then devolve into something close to chaos.

The work itself has existed in many different forms. It began as a program that I could only run myself. Later I turned it into a Web-based project that let viewers upload their own music. Watching people upload works was fascinating: viewers often tried "extreme" music (the most atonal, the noisiest, the silliest top-40 tunes) to stretch the visualization. Many people are startled when they look at visualizations of "low-culture" music, such as a Led Zeppelin song, because the diagrams are so complex. So the artwork is a good anti-snobbery machine.

Finally, I created prints of some of my favorite diagrams. Most of my work has been purely screen-based, so it was a bit of adventure to work on paper. The level of detail provided by print is wonderful. To see all the different scales of structure at once is beautiful, I think.

I'm currently working on a more dynamic version that will include temporal aspects. My goal is to weave together more closely the spatial rhythms with the actual rhythm of the underlying music.

Why did you create *Shape of Song*?

This project was a personal exploration of the nature of music, balance, and the translation between eye and ear. Music visualization has been a subject of interest for centuries, which is one of the appeals of working on it: you have the sense that you are part of history.

I wanted to understand some of the symmetries found in music. Much of music visualization is aimed at literal translations of notes and rhythms into color and animation. Something more abstract appealed to me, and I pursued a representation of the overall musical form instead.

Although the images created in The Shape of Song *are far from a literal translation of the music, the arc-based diagrams came closer than anything else to expressing the mystery and beauty I feel when listening to the underlying compositions.*

Visualization of Vivaldi's Concerto No. 3 in F major, Op. 8, RV 293, "L'autunno." Image courtesy of the artist.

Why did you write your own software tools?

I had to; they didn't exist yet!

A more nuanced answer includes the fact that I actually used a great deal of existing work when writing the code. The method I used to analyze the music is a standard structure in computer science known as a suffix tree. (Suffix trees work by turning sequences into trees and are traditionally used for rapid searching of text. I am always happy when a piece of computer science, designed for mundane purposes, turns out to be useful in an artwork.) During the course of the project I also used a variety of libraries: some provided by Sun, the developers of the Java programming language, for graphics; and some written by others for writing graphics files and reading scores in the "MIDI" music format.

The use of the MIDI format is a quirk of the piece: more common formats, such as MP3, are harder for my algorithms to handle. (You can think of MIDI as analogous to vector graphics, while MP3s are like JPEGs; if you're trying to find simple patterns like squares or circles, it's far easier with a vector format!) At the time I first started working on the piece, MIDI files were extremely common, but they are becoming more and more rare. That raises some questions about the longevity of the piece, but perhaps by the time MIDI is obsolete someone will develop a reliable algorithm for translating MP3s into musical scores.

To sum up, while the goal of my code was original, most of the computer's time is spent in algorithms or code developed by others.

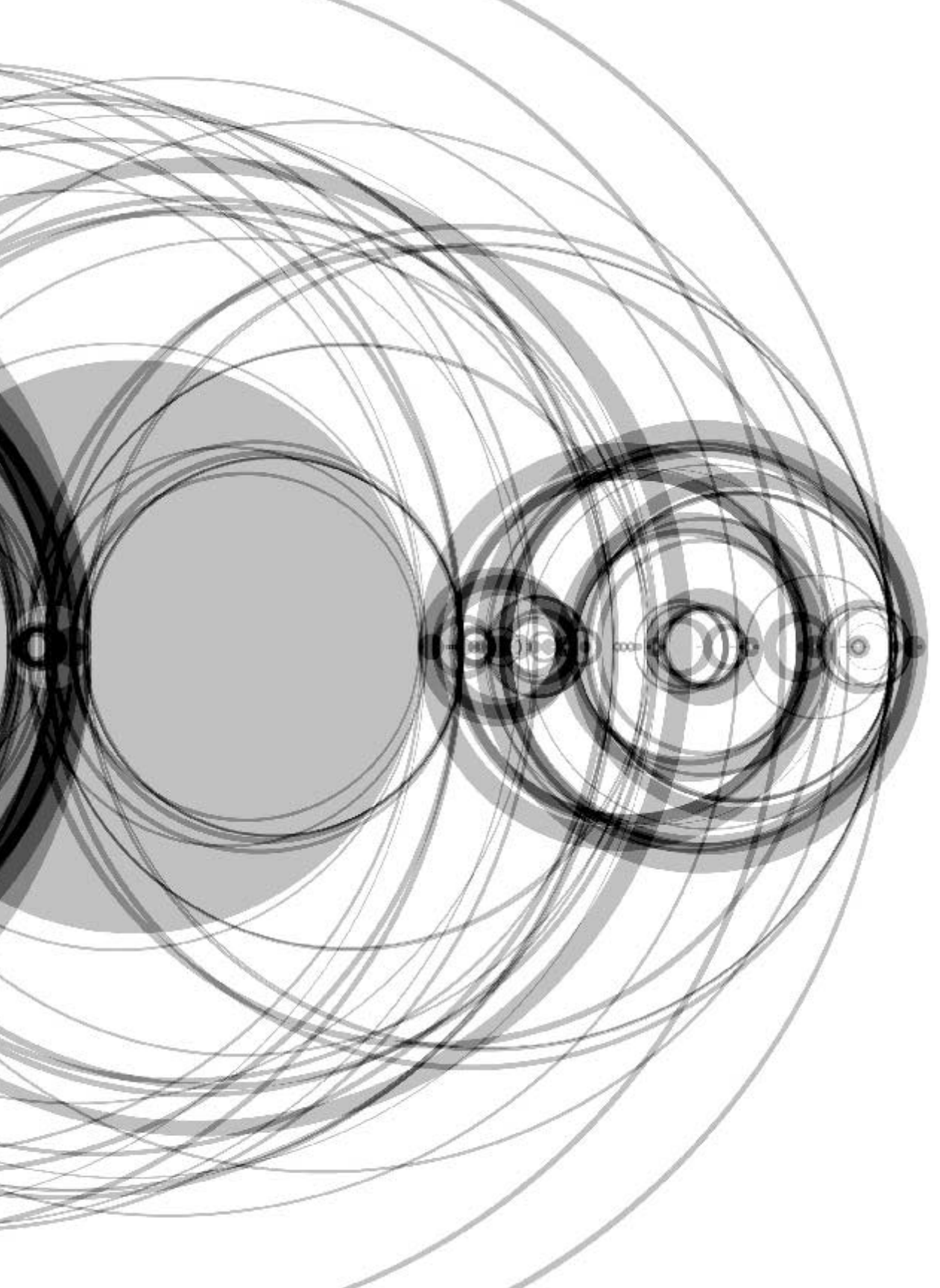
Why do you choose to work with software?

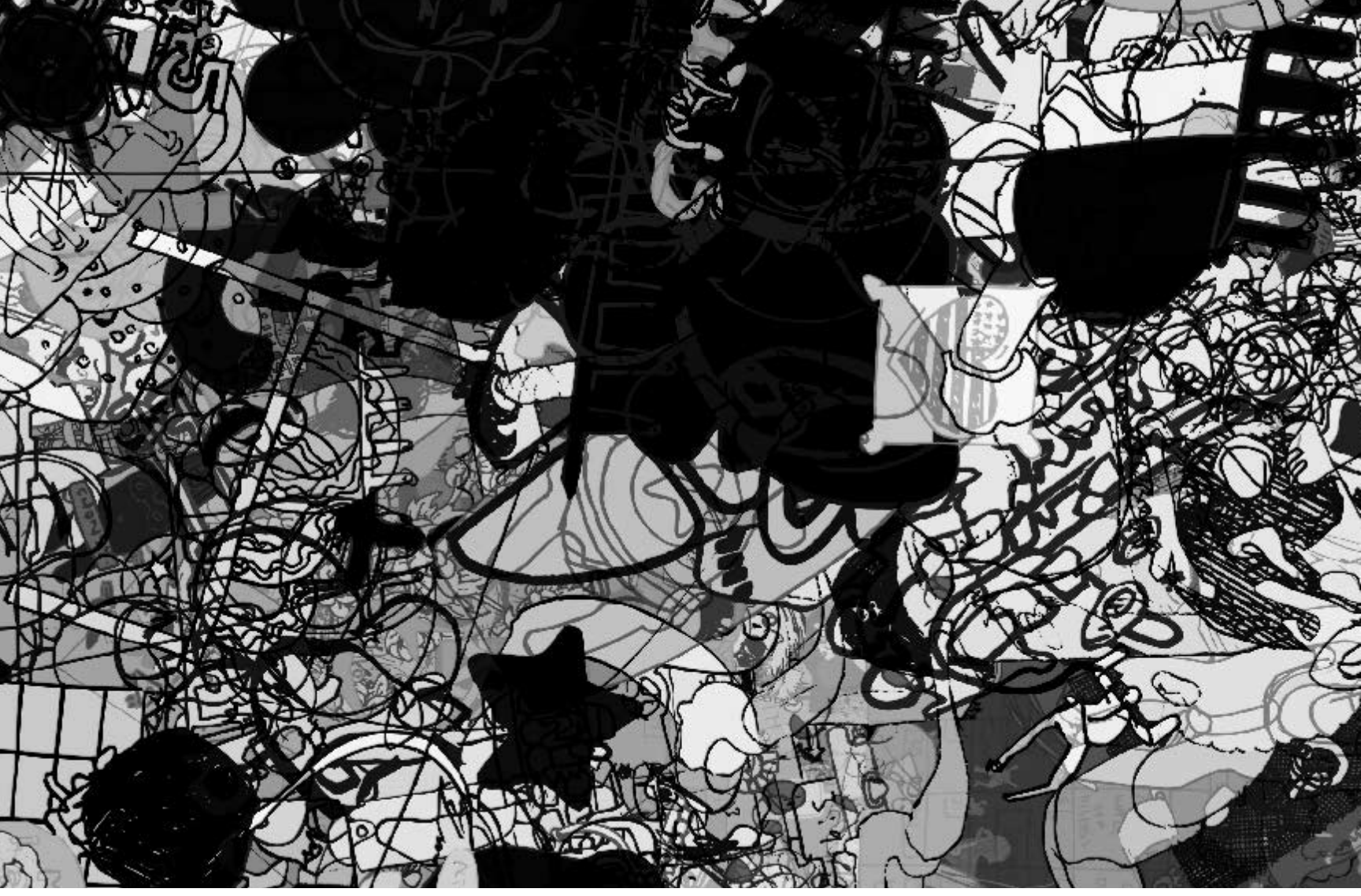
Software is the best way I've found to express myself. When I work in other media, the results somehow always seem worse in reality than in my head. The software I create, however, has a magical quality: it ends up being better than what I originally imagined.

To put it more metaphorically: when I create art, I feel like I am in conversation with the artwork. If I sketch or write, it's like talking to a caustic debater, exposing all the flaws in my thinking. Valuable, perhaps, but also discouraging! When I write programs, I have the opposite feeling: that I am talking with a sympathetic and brilliant partner who helps me organize my thoughts and points out connections I hadn't seen myself.

A second attraction of software art is that it is a new, growing field. There is a kind of energy associated with beginnings that I love. Each new piece seems like it's pointing out new directions, and there's a feeling that you're in a group of settlers on a frontier. As with the American frontier, some people settle down and found new cities, some people keep finding new paths, and some discover gold mines.

Visualization of Vivaldi's Concerto No. 3 in F major, Op. 8, RV 293, "L'autunno." Image courtesy of the artist.





Detail of *Untitled 4*, 2005. Image courtesy of the artist and bitforms gallery, nyc.

The Objectivity Engine *(Interview with James Paterson)*

Creator	James Paterson
Year	2000–present
Medium	Software, Prints
Software	Flash
URL	www.presstube.com

What is *The Objectivity Engine*?

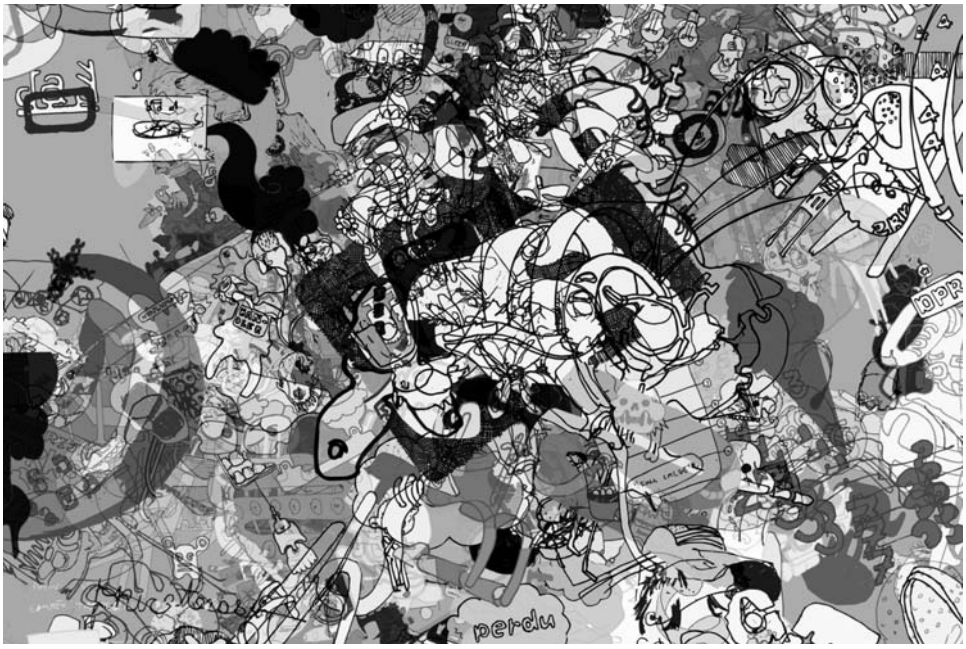
The Objectivity Engine *is a system that I developed to help me make composite images out of my sketchbook drawings. The system has two parts: an ever-expanding library of sketchbook drawings, and an arrangement program that pulls drawings from that library and manipulates them to form the composite images. I have been working on the project since 2000.*

The process of drawing in a sketchbook is a lot like writing a diary for me: whatever is going on in my life ends up in there one way or another. Sometimes I am creating work that I am proud of and other times I might just be scratching down a reminder to buy more pita bread at the supermarket or trying to figure out how much income tax I'm going to pay that year. The drawing library includes all of it. I am not interested in only entering drawings that I like; I include the good and the bad so that the contents reflect my life more naturally.

The content library consists of about 4,000 sketchbook drawings right now but is always growing as I continue to fill books. The drawings are entered chronologically, and looking at all of them in this organized master library helps me to see where I am coming from and where I am going. It is a bit like looking at the rings on a tree trunk. I can see how I was doing at one point or another. I plan on continuing to expand this library as long as I can. Ideally I would like to keep adding to it for the rest of my life.

The arrangement program is a set of algorithms that are in charge of putting together composites. It controls the amount of images used, the vintage of the images, the coloring system, methods of distribution, the motion control system (when it is outputting animation), the scale, rotation, speed, etc. It's like casting a net into the history of my drawing over the past five years. I never know what strange and potentially embarrassing combinations I will reel in. Sometimes seeing the results gives me ideas for new drawings, sometimes I find a composition that I may want to manipulate manually later, and occasionally the program will spit out a composition that I am really happy with and can just save and use as is. A lot of the time it produces complete crap!

I am always adding new features to the arrangement program as I think of them, and about once a year I rewrite the whole thing from scratch. I do this so that I remember how it all works, and so that new components that I am adding as I go along can be properly integrated. Just as I fancy the idea of continuing to add to the library of drawings for the rest of my life, I also like the idea of adding to the capability of the arrangement system over a long period of time and seeing where it leads me.



Untitled 3, 2005. Digital Lambda print. 40" × 60". Image courtesy of the artist and bitforms gallery, nyc.

Why did you create *The Objectivity Engine*?

I have never been very good at putting disparate ideas and images together to form larger, more coherent finished works, so I decided to create a system that would get me started down this road. It is an assistant that helps me make decisions, and it does a lot of tedious and repetitive work for me very quickly.

What software tools were used?

Photoshop was used to scan and clean up the sketchbook drawings. Streamline was then used to convert the raster scans to vector format. The vector files were brought into Flash. Flash was used to contain the drawing library, write the arrangement software, and export the results. Director was used to export animated results. Photoshop, Illustrator, and After Effects were used to put finishing touches on the raw exports and prep them for print and DVD.

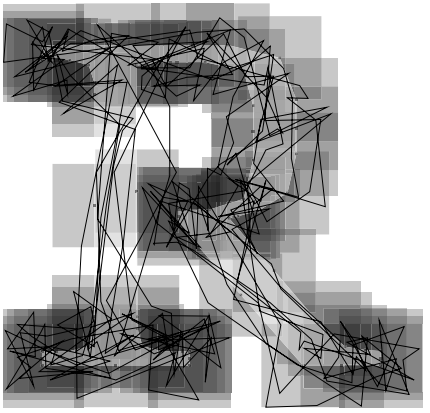
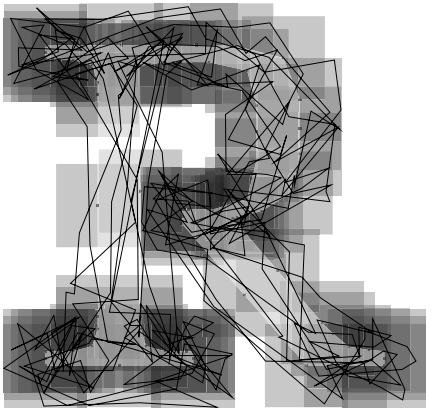
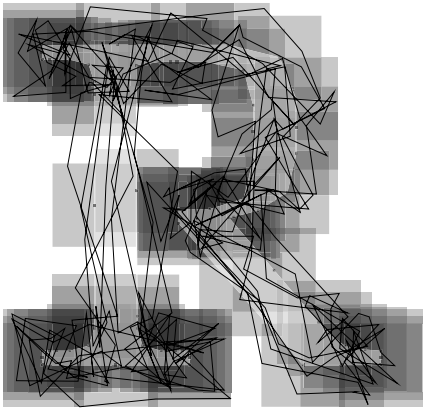
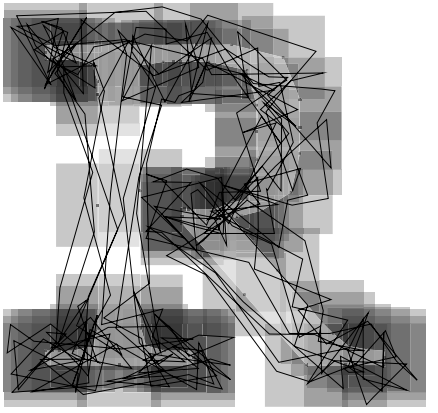
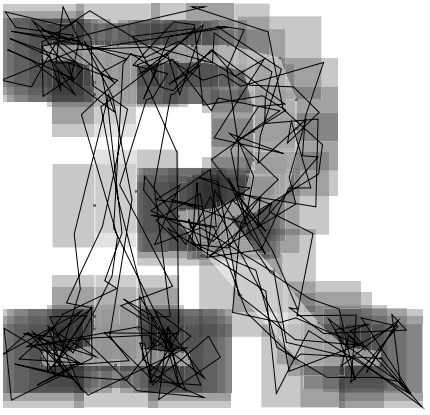
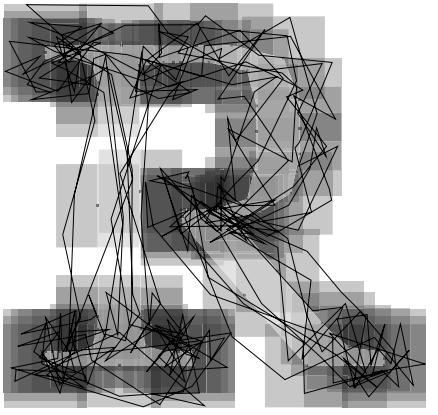
Why did you use these tools?

They just seemed like the best tools I could find for each respective job. I used Flash/ ActionScript as my main platform because I have been using it for a long time and I use it to create all of my other work. This way I can mix elements of this project with many other works that have nothing to do with it and vice versa. Also, Flash is a vector-based environment, which lends itself well to the aims of the project.

Why do you choose to work with software?

I just stumbled across Flash in 1997 and started playing with it and then never stopped. I had been drawing for quite some time before that, and after a few years my drawing processes merged with my software processes and the two have been pretty much inseparable ever since.

Untitled 5, 2005. Digital Lambda print. 40" × 60". Image courtesy of the artist and bitforms gallery, nyc.
Untitled 10, 2005. Digital Lambda print. 40" × 60". Image courtesy of the artist and bitforms gallery, nyc.

RandomFont Beowolf *(Interview with Erik van Blokland)*

Creators	Just van Rossum and Erik van Blokland (LettError)
Year	1990
Medium	Typeface
Software	PostScript Type 3 font
URL	www.letterror.com/foundry/beowolf

What is RandomFont Beowolf?

In 1989, after finishing the graphic and typographic design course at the Royal Academy of Arts in The Hague, Netherlands, Just and I were both experimenting with PostScript, a powerful programming language designed for graphics. The only machines we had access to that were capable of executing PostScript programs were laser printers, so we went through a lot of paper. We figured it should be possible to build a font with random functions applied to the letterforms when it prints. We made a test font containing one randomizing square to prove the concept. Later we used a typeface I had drawn in school, which after some iterations became Beowolf.

A font in PostScript in its most basic form is a dictionary with some standard entries and drawing instructions for each letter. Fonts like this were called "Type 3," and as long as you managed to get the file in the printer, the letters could use all the functionality of the PostScript language. So that's what we built. Another PostScript font format, "Type 1," was more compact and offered faster processing and hinting, but it was proprietary, encrypted and required secret tools to generate. In the summer of 1990 Adobe published the specifications of Type 1 but unfortunately the increased speed of printing with Type 1 fonts came at a price of a very limited instruction set.

Later we built "piggyback fonts," which incorporated both Type 1 and Type 3 formats. Fonts like this consisted of several Type 1 fonts and a special Type 3 font to contain logic to switch glyphs and fonts while rendering a text. The fonts were all bundled together in a single file, then the Macintosh printer driver would just download the whole thing, and all fonts would make it to the printer. It was a wonderful hack.

The demands of the graphic design workflow made it increasingly difficult to deploy Type 3 formats, and we stopped shipping them. We've always made a case for fonts with executable code; typography is a complex field which can benefit from programming on a font level. But we can't expect the entire digital design industry to accommodate our whims.

The current OpenType font format (developed by Adobe and Microsoft) actually contains ways to define rules for contextual substitution[1] and positioning of glyphs. Though nothing like a fully featured programming language, it's an improvement and fun to develop for. We have an OpenType font with a decent simulation of a RandomFont, much like our piggyback fonts. Visually it reconstructs the broken, edgy style Beowolf had, but conceptually these OpenType Beowolves have very little to do with the original one.

Why did you create RandomFont Beowolf?

Curiosity. We were both trained as type designers, and we were interested in computers and programming. At that time the fields of design and digital technology didn't really overlap. After the first versions we started thinking about the context and implications of

Randomized Beowolf letters created with DrawBot. Image courtesy of Just van Rossum and Erik van Blokland.

randomization. Beowolf became an example of what digital type could be: not that the random aesthetic itself was so appealing, but it was proof that fonts were no longer physical objects but instructions. It also showed us that code and design can merge into a single process, a single object. Code has a major influence on design, and I think it is too important to leave it to anonymous engineers.

What software tools were used?

The first PostScript tests were written in a text editor. Then we took simple PostScript Type 3 fonts generated by Ikarus M (the first version, written by Petr van Blokland) and edited those. These fonts had simple, readable text instructions and absolute coordinates that were easy to modify. Later on with some help from AltSys's Jim Von Ehr we moved on to a more complex variation of Type 3 PostScript, editing in ResEdit. The last incarnation of the RandomFonts was piggybacking on a Type 1 font so that there would be some sort of (nonrandom) preview when used in a layout program; then later in the printer the Type 3 randomizing version would kick in and do the work. The piggyback fonts were also made with ResEdit.

Why did you use these tools?

Type design is a small field, so there aren't many developers interested in writing tools. AltSys wrote Fontographer, at the time the popular choice for editing fonts, but it didn't allow the things we had in mind. Petr van Blokland and David Berlow convinced Jim Von Ehr to give them access to the Fontographer source code and Petr started experimenting with adding a layer using awk, a programming language for processing text data. Just suggested using a new programming language that his brother had invented called Python. Python and Fontographer became RoboFog and gained a small but dedicated group of users. Python is fast to develop in, which allows a regular iterative design process. After the first time you write a program you know how it should have been done and you can afford to start over and do it again, improving the understanding of the problem.

When Mac OS X was released we couldn't port RoboFog because the code base was so old. The FontLab font development software had added Python scripting to their font editor, so we started to work with that instead. We started reshaping their API into ours and this grew into the RoboFab library, an object model and API built on top of FontLab's font and glyph objects. RoboFab now also has an implementation that works independently of FontLab.

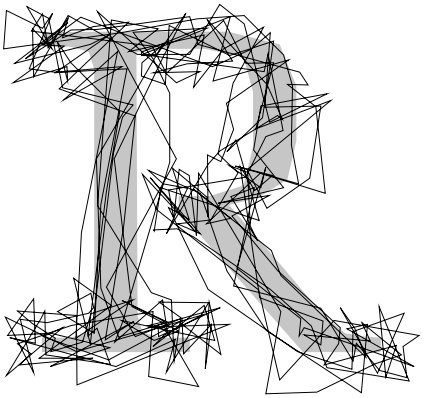
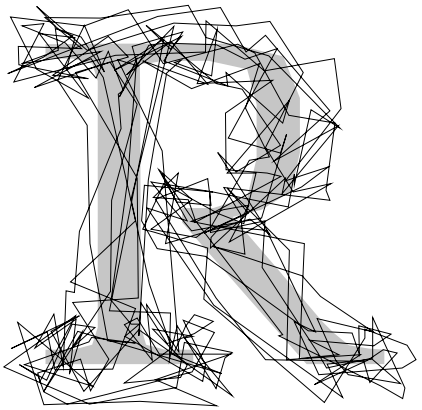
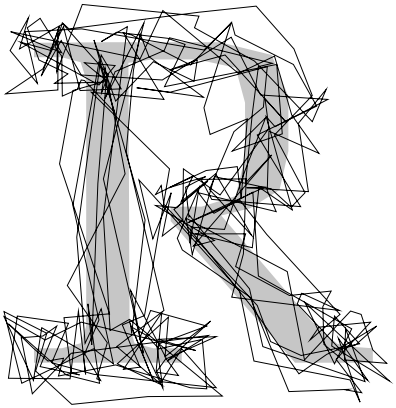
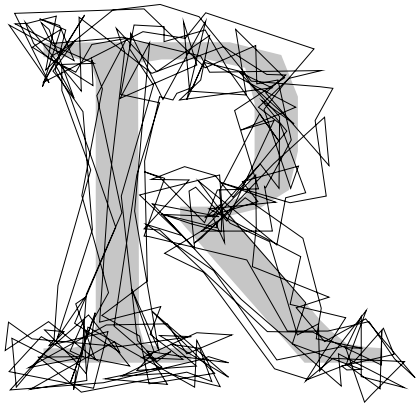
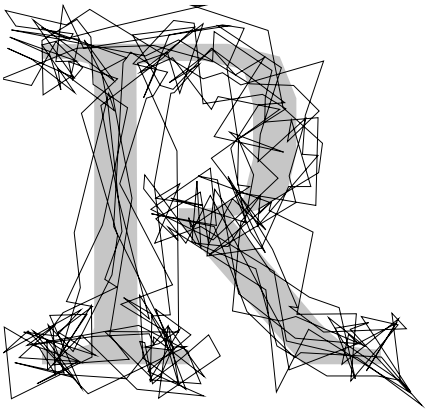
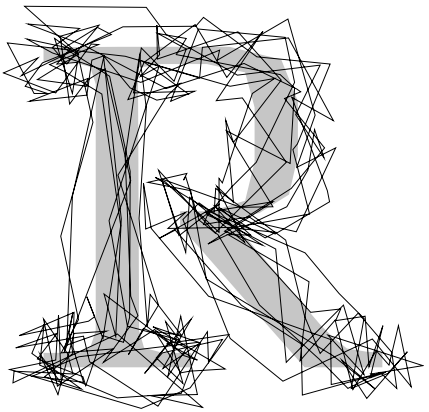
Why do you choose to work with software?

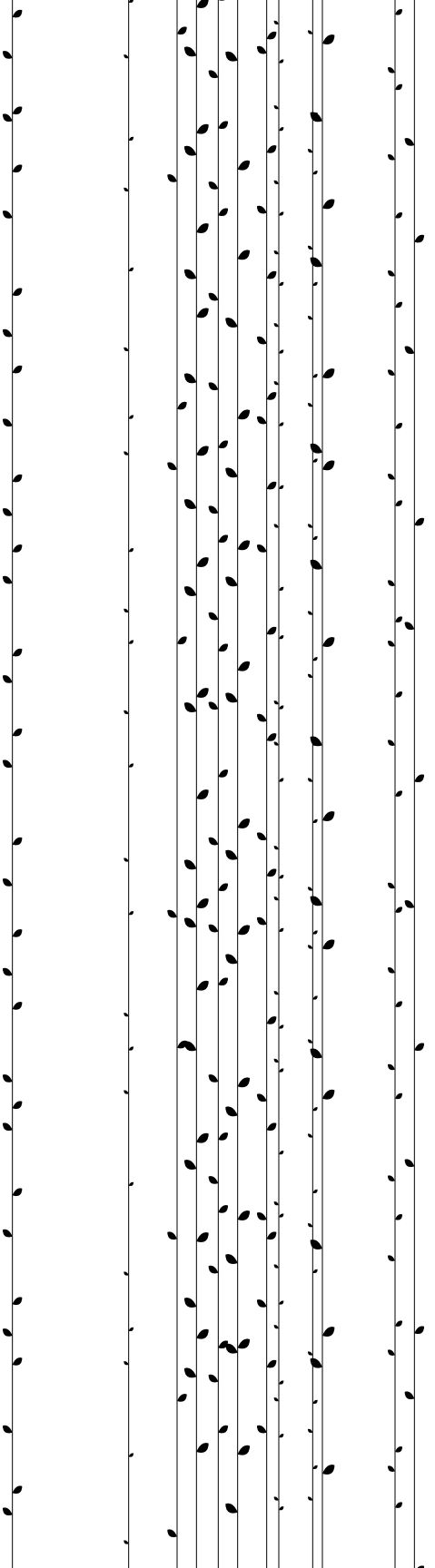
In any creative discipline, the tools influence the process and, indirectly, the results. We try to be aware of this influence, and if it is something we don't like we try to change it. Every application makes certain things easy and others more difficult. This directs the average design project towards the things that are easy, even though other ideas might be more relevant.

Writing your own tools makes the ideas direct the development of software, rather than the other way round. Writing code is also an attractive process in itself. Analyzing problems, breaking them down into ideas that can be coded, and discovering alternative new ways to solve known problems is more universal than just the original design task.

Notes

1. Contextual substitution is the process by which letters are changed based on the letters around them. For instance, an *f* and *i* might be joined into a single glyph, a ligature that looks like fi.



Structure 2: Continuous

The unit introduces programs that run continuously and explains how to control their speed.

Syntax introduced:
`draw(), frameRate(), frameCount, setup(), noLoop()`

All the programs in preceding units run their code once and the program stops. Programs that animate or respond to live information must run continuously. Continuously running programs can create animation or use the mouse and keyboard for input.

Continuous evaluation

Programs that run continuously must include a `draw()` function. The code inside a `draw()` block runs in a loop until the the stop button is pressed or the window is closed. A program can have only *one* `draw()`. Each time the `draw()` function finishes, it draws a new frame to the display window and then starts running the block again from the first line.

By default, frames are drawn to the screen at 60 frames per second (fps). The `frameRate()` function changes and controls the number of frames displayed each second. The program will always attempt to run at the speed set by the parameter to the `frameRate()` function, but sometimes the ambitions of the programmer exceed the speed of the computer. The `frameRate()` function controls only the maximum frame rate—it can not speed up a program that runs slowly because of equipment limitations.

The `frameCount` variable always contains the number of frames displayed since the program started. A program with `draw()` keeps displaying frames (1, 2, 3, 4, 5, . . .) until it is stopped, the computer is shut down, or the power goes out.

20-01
```
// Prints each frame number to the console
void draw() {
  println(frameCount);
}
```

20-02
```
// Runs at around 4 fps, prints each frame number to the console
void draw() {
  frameRate(4);
  println(frameCount);
}
```

Changing visual elements from frame to frame creates animation. For example, changing the position of a line each frame will cause it to move:

20-03

```
float y = 0.0;

void draw() {
  frameRate(30);
  line(0, y, 100, y);
  y = y + 0.5;
}
```

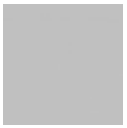
When this code runs, the variables are replaced with their current values and the statements are run in this order:

```
float y = 0.0
frameRate(30)                    Enter draw()
line(0, 0.0, 100, 0.0)
y = 0.5
frameRate(30)                    Enter draw() for the second time
line(0, 0.5, 100, 0.5)
y = 1.0
frameRate(30)                    Enter draw() for the third time
line(0, 1.0, 100, 1.0)
y = 1.5
Etc...
```

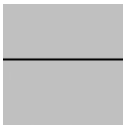
The variable `y` must be declared outside `draw()` for this program to move the line each frame. If the variable is declared inside `draw()`, it will be re-created each time the `draw()` block is run and reassigned to the same value, placing the line in the same position.

The background of the display window does not refresh automatically, so lines will simply accumulate. To clear the display window at each frame, insert a `background()` function at the beginning of the `draw()` function. The `background()` function fills the entire display window with the specified color. It overwrites every pixel in the display window each time it is run. If the `background()` is not placed at the top of `draw()`, it will cover any element drawn earlier.

y=10



y=46



y=84

20-04

```
float y = 0.0;

void draw() {
  frameRate(30);
  background(204);
  y = y + 0.5;
  line(0, y, 100, y);
}
```

The variable that controls the line position can be used for other purposes. In the next example, it's also used to set the color of the background.

y=20

y=49

y=88

20-05

```
float y = 0.0;

void draw() {
  frameRate(30);
  background(y * 2.5);
  y = y + 0.5;
  line(0, y, 100, y);
}
```

After a few seconds, the line moves off the bottom edge of the display window. An `if` structure can reset the value of the variable so the line position is set back to the top:

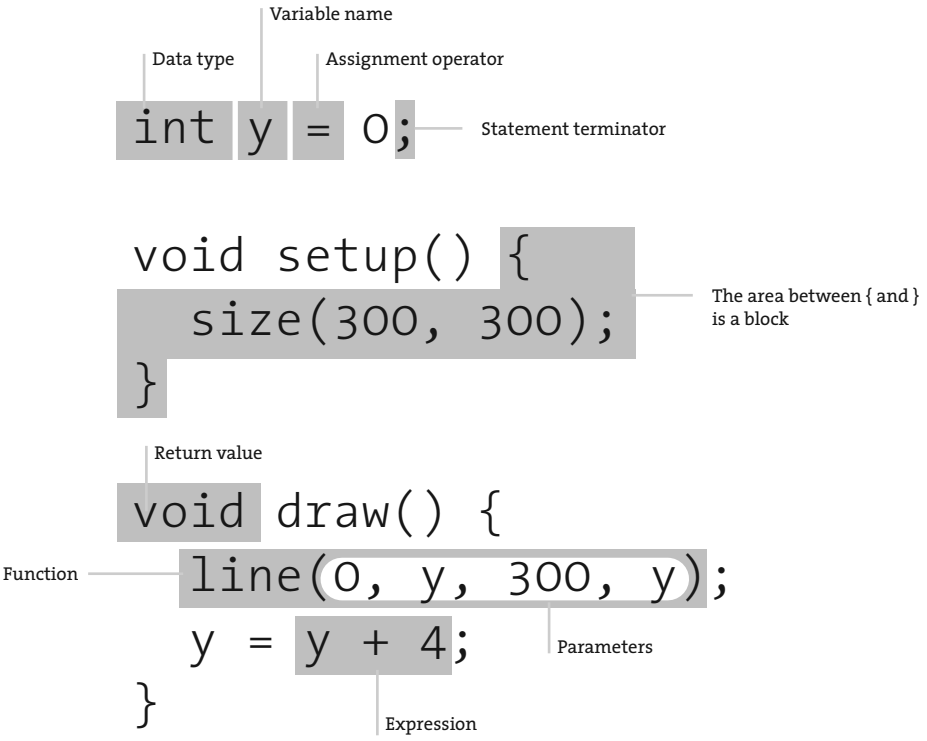
y=30

y=88

y=19

20-06

```
float y = 0.0;

void draw() {
  frameRate(30);
  background(y * 2.5);
  y = y + 0.5;
  line(0, y, 100, y);
  if (y > 100) {
    y = 0;
  }
}
```



Anatomy of a program 2

Each program can have only one `setup()` and one `draw()`. When the program starts, the code outside of `setup()` and `draw()` is run. Next, the code inside the `setup()` block is run once. After that, the code inside the `draw()` block is run continuously until the program is stopped. Because the variable `y` is declared outside of `setup()` and `draw()`, it's a global variable and can be accessed and assigned anywhere within the program.

Controlling the flow

Some functions need to be run once, rather than every frame. The `setup()` function is run before `draw()` so that functions like `size()` or `loadImage()` aren't rerun on each frame. When a program is run, the code outside `setup()` and `draw()` is handled first, then code inside `setup()` is run once, and finally the code inside `draw()` is run in a continuous loop from top to bottom. In the following example, the size, antialiasing setting, and fill value don't change, so they are included in `setup()`.

y=67

y=108

y=12

20-07

```
float y = 0.0;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
}

void draw() {
  background(204);
  ellipse(50, y, 70, 70);
  y += 0.5;
  if (y > 150) {
    y = -50.0;
  }
}
```

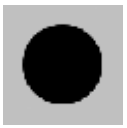
When this code runs, the variables are replaced with their current values and the statements are run in this order:

```
float y = 0.0
size(100, 100)                    Enter setup()
smooth()
fill(0)
background(204)                   Enter draw()
ellipse(50, 0.0, 70, 70)
y = 0.5
background(204)                   Enter draw() for the second time
ellipse(50, 0.5, 70, 70)
y = 1.0
background(204)                   Enter draw() for the third time
ellipse(50, 1.0, 70, 70)
y = 1.5
Etc....
```

When the value of y becomes greater than 150, the code in the `if` structure block sets the value to –50.

Variables that change with each iteration of `draw()` must be declared outside of both `setup()` and `draw()`. If the variable y in the preceding example were declared in `draw()`, it would be reassigned to 0.0 each time. The only statements that should occur outside `setup()` and `draw()` are variable declarations and assignments. Running functions outside `setup()` and `draw()` will cause an error.

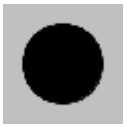
If a program only draws one frame, it can be written entirely inside `setup()`. The only difference between `setup()` and `draw()` is that `setup()` is run once before `draw()` starts looping, therefore shapes drawn within `setup()` will appear in the display window.



20-08

```
void setup() {
  size(100, 100);
  smooth();
  fill(0);
  ellipse(50, 50, 66, 66);
}
```

Using the `noLoop()` function stops `draw()` from looping and can be used as another way to draw only one frame. This example is similar to the previous one, but runs the code in `setup()` once and then runs the code in `draw()` only once because `noLoop()` is called in `setup()`.



20-09

```
void setup() {
  size(100, 100);
  smooth();
  fill(0);
  noLoop();
}

void draw() {
  ellipse(50, 50, 66, 66);
}
```

Variable scope

When `setup()` and `draw()` are added to a program, it becomes necessary to think about where variables are declared and assigned. The location of a variable declaration determines its *scope*—where it can be accessed within the program. The rule for variable scope is stated simply: variables declared inside any block can be accessed only inside their own block and inside any blocks enclosed within their block. Variables declared at the base level of the program—the same level as `setup()` and `draw()`—

can be accessed everywhere within the program. Variables declared within `setup()` can be accessed only within the `setup()` block. Variables declared within `draw()` can be accessed only within the `draw()` block. The scope of a variable declared within a block, called a local variable, extends only to the end of the block.

20-10

```
int d = 51;          // Variable d can be used everywhere

void setup() {
  size(100, 100);
  int val = d * 2;   // Local variable val can only be used in setup()
  fill(val);
}

void draw() {
  int y = 60;        // Local variable y can only be used in draw()
  line(0, y, d, y);
  y -= 25;
  line(0, y, d, y);
}
```

When a variable is created within a block, it is destroyed when the program leaves the block. For example, if a new variable is created inside an `if` block, it can be used within but cannot be accessed outside the block. If a new variable is used to iterate through a `for` structure, it can be used within but cannot be accessed outside the block.

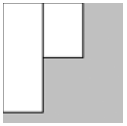
20-11

```
void draw() {
  int d = 80;    // This variable can be used everywhere in draw()
  if (d > 50) {
    int x = 10;  // This variable can be used only in this if block
    line(x, 40, x+d, 40);
  }
  line(0, 50, d, 50);
  line(x, 60, x+d, 60);  // ERROR! x can't be read outside block
}
```

20-12

```
void draw() {
  for (int y = 20; y < 80; y += 6) {  // The variable y can be used
    line(20, y, 50, y);               // only within the for block
  }
  line(y, 0, y, 100);  // ERROR! y can't be accessed outside for
}
```

Variable scope makes it possible to have more than one variable in a program with the same name. It's common to use the same variable name for iterating over multiple `for` structures in one program, but in general, having more than one variable with the same name is not recommended. The following example demonstrates this potentially confusing case.



```
int d = 45;             // Assign 45 to variable d

void setup() {
  size(100, 100);
  int d = 90;           // Assign 90 to local variable d
  rect(0, 0, 33, d);    // Use local d with value 90
}

void draw() {
  rect(33, 0, 33, d);   // Use d with value 45
}
```

20-13

A variable inside a block with the same name as a variable outside the block is a common mistake that can be confusing to debug.

Exercises

1. *Make a program run at four frames per second and display the current frame count to the console with* `println()`.
2. *Move a shape from left to right across the screen. When it moves off the right edge, return it to the left.*
3. *Utilize* `noLoop()` *to make a program run its* `draw()` *only one time.*

Structure 3: Functions

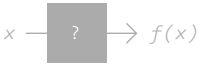
This unit introduces basic concepts and syntax for writing functions.

Syntax introduced:
`void, return`

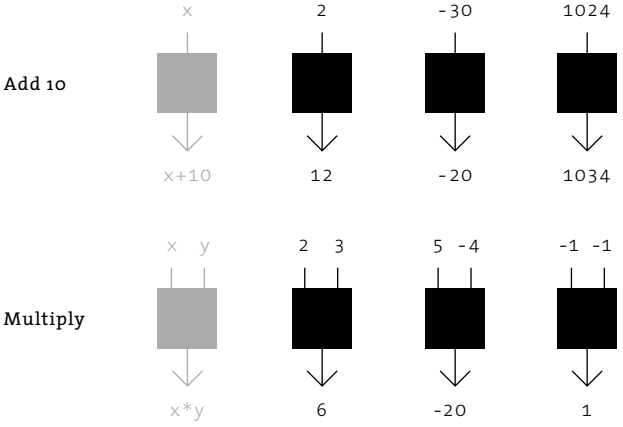
A function is a self-contained programming module. You've been using the functions included with Processing such as `size()`, `line()`, `stroke()`, and `translate()` to write your programs, but it's also possible to write your own functions that make a program modular. Functions make redundant code more concise by extracting the common elements and making them into code blocks that can be run many times within the program. This makes the code easier to read and update and reduces the chance of errors.

Functions often have parameters to define their actions. For example, the `line()` function has four parameters that define the position of the two points. Changing the numbers used as parameters changes the position of the line. Functions can operate differently depending on the number of parameters used. For example, a single parameter to the `fill()` function defines a gray value, two parameters define a gray value with transparency, and three parameters define an RGB color.

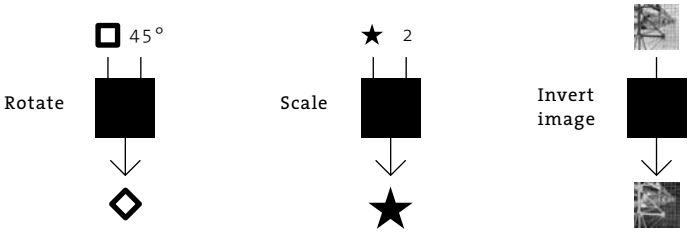
A function can be imagined as a box with mechanisms inside that act on data. There is typically an input into the box and code inside that utilizes the input to produce an output:



For example, a function can be written to add 10 to any number or to multiply two numbers:



The previous function examples are simple, but the concept can be extended to other processes that may be less obvious:



The mathematics used inside functions can be daunting, but the beauty of using functions is that it's not necessary to understand how they work. It's usually enough to know how to use them—to know what the inputs are and how they affect the output. This technique of ignoring the details of a process is called abstraction. It helps place the focus on the overall design of the program rather than the details.

Abstraction

In the terminology of software, the word *abstraction* has a different meaning from how it's used to refer to drawings and paintings. It refers to hiding details in order to focus on the result. The interface of the wheel and pedals in a car allows the driver to ignore details of the car's operation such as firing pistons and the flow of gasoline. The only understanding required by the person driving is that the steering wheel turns the vehicle left and right, the accelerator speeds it up, and the brake slows it down. Ignoring the minute details of the engine allows the driver to maintain focus on the task at hand. The mind need not be cluttered with thoughts about the details of execution.

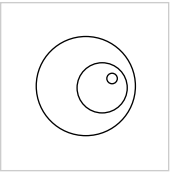
The idea of abstraction can also be discussed in relation to the human body. For example, we can control our breathing, but we usually breathe involuntarily, without conscious thought. Imagine if we had to directly control every aspect of our body. Having to continually control the beating of the heart, the release of chemicals, and the firing of neurons would make reading books and writing software impossible. The brain abstracts the basic functions of maintaining the body so our conscious minds can consider other aspects of life.

The idea of abstraction is essential to writing software. In Processing, drawing functions such as `line()`, `ellipse()`, and `fill()` obscure the complexity of their actions so that the author can focus on results rather than implementation. If you want to draw a line, you probably want to think only about its position, thickness, and color, and you don't want to think about the many lines of code that run behind the scenes to convert the line into a sequence of pixels.

Creating functions

Before explaining in detail how to write your own functions, we'll first look at an example of why you might want to do so. The following examples show how to make a program shorter and more modular by adding a function. This makes the code easier to read, modify, and expand.

It's common to draw the same shape to the screen many times. We've created the shape you see below on the left, and now we want to draw it to the screen in the pattern on the right:



We start by drawing it once, to make sure our code is working.

21-01

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  fill(255);
  ellipse(50, 50, 60, 60);     // White circle
  fill(0);
  ellipse(50+10, 50, 30, 30);  // Black circle
  fill(255);
  ellipse(50+16, 45, 6, 6);    // Small, white circle
}
```

The previous program presents a sensible way to draw the shape once, but when another shape is added, we see a trend that continues for each additional shape. Adding a second shape inside `draw()` doubles the amount of code. Because it takes 6 lines to draw each shape, we now have 12 lines. Drawing our desired pattern that uses 6 shapes will require 36 lines of code. Imagine if we wanted to draw 30 eyes—the code inside `draw()` would bloat to 180 lines.

21-02

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  // Right shape
  fill(255);
  ellipse(65, 44, 60, 60);
  fill(0);
  ellipse(75, 44, 30, 30);
  fill(255);
  ellipse(81, 39, 6, 6);
  // Left shape
  fill(255);
  ellipse(20, 50, 60, 60);
  fill(0);
  ellipse(30, 50, 30, 30);
  fill(255);
  ellipse(36, 45, 6, 6);
}
```

Because the shapes are identical, a function can be written for drawing them. The function introduced in the next example has two inputs that set the x-coordinate and y-coordinate. The lines of code inside the function render the elements for one shape.

21-03

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  eye(65, 44);
  eye(20, 50);
}

void eye(int x, int y) {
  fill(255);
  ellipse(x, y, 60, 60);
  fill(0);
```

```
    ellipse(x+10, y, 30, 30);
    fill(255);
    ellipse(x+16, y-5, 6, 6);
  }
```

21-03 cont.

The function is 8 lines of code, but it only has to be written once. The code in the function runs each time it is referenced in `draw()`. Using this strategy, it would be possible to draw 30 eyes with only 38 lines of code.

A closer look at the flow of this program reveals how functions work and affect the program flow. Each time the function is used within `draw()`, the 6 lines of code inside the function block are run. The normal flow of the program is diverted by the function call, the code inside the function is run, and then the program returns to read the next line in `draw()`. Because `noLoop()` is used inside `setup()`, the lines of code in `draw()` only run once.

```
size(100, 100)                    Start with code in setup()
noStroke()
smooth()
noLoop()
fill(255)                         Enter draw(), divert to the eye function
ellipse(65, 44, 60, 60)
fill(0)
ellipse(75, 44, 30, 30)
fill(255)
ellipse(81, 39, 6, 6)             Back to draw(), divert to the eye function a second time
fill(255)
ellipse(20, 50, 60, 60)
fill(0)
ellipse(30, 50, 30, 30)
fill(255)
ellipse(36, 45, 6, 6)             Program ends
```

Now that the function is working, it can be used each time we want to draw that shape. If we want to use the shape in another program, we can copy and paste the function. We no longer need to think about how the shape is being drawn or what each line of code inside the function does. We only need to remember how to control its position with the two parameters.

21-04

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  eye(65, 44);
  eye(20, 50);
  eye(65, 74);
  eye(20, 80);
  eye(65, 104);
  eye(20, 110);
}

void eye(int x, int y) {
  fill(255);
  ellipse(x, y, 60, 60);
  fill(0);
  ellipse(x+10, y, 30, 30);
  fill(255);
  ellipse(x+16, y-5, 6, 6);
}
```

To write a function, start with a clear idea about what the function will do. Does it draw a specific shape? Calculate a number? Filter an image? After you know what the function will do, think about the parameters and the data type for each. Have a goal and break the goal into small steps.

In the following example, we first put together a program to explore some of the details of the function before writing it. Then, we start to build the function, adding one parameter at a time and testing the code at each step.

21-05

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  // Draw thick, light gray X
  stroke(160);
  strokeWeight(20);
  line(0, 5, 60, 65);
```

```
  line(60, 5, 0, 65);
  // Draw medium, black X
  stroke(0);
  strokeWeight(10);
  line(30, 20, 90, 80);
  line(90, 20, 30, 80);
  // Draw thin, white X
  stroke(255);
  strokeWeight(2);
  line(20, 38, 80, 98);
  line(80, 38, 20, 98);
}
```

21-05 cont.

To write a function to draw the three X's in the previous example, first write a function to draw just one. We named the function `drawX()` to make its purpose clear. Inside, we have written code that draws a light gray X in the upper-left corner. Because this function has no parameters, it will always draw the same X each time its code is run. The keyword `void` appears before the function's name, which means the function does not return a value.

21-06

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX();
}

void drawX() {
  // Draw thick, light gray X
  stroke(160);
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}
```

To draw the X differently, add a parameter. In the next example the `gray` parameter variable has been added to the function to control the gray value of the X. The parameter variable must include its type and its name. When the function is called from within `draw()`, the value within the parentheses to the right of the function name is assigned to `gray`. In this example, the value 0 is assigned to `gray`, so the stroke is set to black.

21-07

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(0);  // Passes 0 to drawX(), runs drawX()
}

void drawX(int gray) {  // Declares and assigns gray
  stroke(gray);         // Uses gray to set the stroke
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}
```

A function can have more than one parameter. Each parameter for the function must be placed between the parentheses after the function name, each must state its data type, and the parameters must be separated by commas. In this example, the additional parameter `weight` is added to control the thickness of the line.

21-08

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(0, 30);  // Passes values to drawX(), runs drawX()
}

void drawX(int gray, int weight) {
  stroke(gray);
  strokeWeight(weight);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}
```

The next example extends `drawX()` to three additional parameters that control the position and size of the X drawn with the function.

21-09

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(0, 30, 40, 30, 36);
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

By carefully building our function one step at a time, we have reached the original goal of writing a general function for drawing the three X's in code 21-05 (p. 186).

21-10

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(160, 20, 0, 5, 60);   // Draw thick, light gray X
  drawX(0, 10, 30, 20, 60);   // Draw medium, black X
  drawX(255, 2, 20, 38, 60);  // Draw thin, white X
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

Now that we have the `drawX()` function, it's possible to write programs that would not be practical without it. For example, putting calls to `drawX()` inside a `for` structure allows for many repetitions. Each X drawn can be different from those previously drawn.

21-11

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  for (int i = 0; i < 20; i++) {
    drawX(200- i*10, (20-i)*2, i, i/2, 70);
  }
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

21-12

```
void setup() {
  size(100, 100);
  smooth();
  noLoop()
}

void draw() {
  for (int i = 0; i < 70; i++) {  // Draw 70 X shapes
    drawX(int(random(255)), int(random(30)),
          int(random(width)), int(random(height)), 100);
  }
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

In the next series of examples, a `leaf()` function is created from code 7-17 (p. 77) to draw a leaf shape, and a `vine()` function is created to arrange a group of leaves onto a line. These examples demonstrate how functions can run inside other functions. The `leaf()` function has four parameters that determine the position, size, and orientation:

`float x`	*X-coordinate*
`float y`	*Y-coordinate*
`float size`	*Width of the leaf in pixels*
`int dir`	*Direction, either 1 (left) or -1 (right)*

This simple program draws one leaf and shows how the parameters affect its attributes.

21-13

```
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  noLoop();
}

void draw() {
  leaf(26, 83, 60, 1);
}

void leaf(int x, int y, int size, int dir) {
  pushMatrix();
  translate(x, y);  // Move to position
  scale(size);      // Scale to size
  beginShape();     // Draw the shape
  vertex(1.0*dir, -0.7);
  bezierVertex(1.0*dir, -0.7, 0.4*dir, -1.0, 0.0, 0.0);
  bezierVertex(0.0, 0.0, 1.0*dir, 0.4, 1.0*dir, -0.7);
  endShape();
  popMatrix();
}
```

The `vine()` function has parameters to set the position, the number of leaves, and the size of each leaf:

`int x`	*X-coordinate*
`int numLeaves`	*Total number of leaves on the vine*
`float leafSize`	*Width of the leaf in pixels*

This function determines the form of the vine by applying a few rules to the parameter values. The code inside `vine()` first draws a white vertical line, then determines the

space between each leaf based on the height of the display window and the total number of leaves. The first leaf is set to draw to the right of the vine, and the `for` structure draws the number of leaves specified by the `numLeaves` parameter. The `x` parameter determines the position, and `leafSize` sets the size of each leaf. The y-coordinate of each leaf is slightly different each time the program is run because of the `random()` function.

21-14

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  vine(33, 9, 16);
}

void vine(int x, int numLeaves, int leafSize ) {
  stroke(255);
  line(x, 0, x, height);
  noStroke();
  int gap = height / numLeaves;
  int direction = 1;
  for (int i = 0; i < numLeaves; i++) {
    int r = int(random(gap));
    leaf(x, gap*i + r, leafSize, direction);
    direction = -direction;
  }
}

// Copy and paste the leaf() function here
```

The `vine()` function was written in steps and was gradually refined to its present code. It could be extended with more parameters to control other aspects of the vine such as the color, or to draw on a curve instead of a straight line. In these examples, the vine function is called from `draw()` and the qualities of the vine are set by different parameters.

Shorter programs aren't the only benefit of using functions, but less code has advantages beyond a reduction in typing. Shorter programs lead to fewer errors—the more lines of code, the more chances for mistakes.

Imagine a novel written as a continuous paragraph without indentations or line breaks. Functions act as paragraphs that make your program easier to read. The practice of reducing complex processes into smaller, easier-to-comprehend units helps structure

ideas. But it's not simply a matter of making lots of functions. Each function should be a unit of code that clearly expresses a single idea, calculation, or unit of form.

In code 21-02, six lines of code are needed to draw each shape. If we wanted to change one small detail—for example, the position of the small white circle in relation to the black circle—the corresponding line would need to be changed several times. If we were drawing nine shapes, nine lines of code would have to be changed. Once a group of code is put into a function, the program is easy to modify because that line of code need only be changed once.

Functions can make programs easier to write because they encourage reusing code. A custom function can be reused in another program. As you write more programs, you'll build a collection of functions that are useful across much of your work. In fact, parts of Processing evolved from function collections that the authors used in their own work.

Function overloading

Multiple functions can have the same name, as long as they have different parameters. Creating different functions with the same name is called function overloading, and it's what allows Processing to have more than one version of functions like `fill()`, `image()`, and `text()`, each with different parameters. For example, the `fill()` function can have one, two, three, or four parameters. Each version of `fill()` sets the fill value for drawing shapes, but the number of parameters determines whether the fill value is gray, color, or includes transparency.

A program can also have two functions with the same number of parameters, but only if the data type for one of the parameters is different. For example, there are three versions of the `fill()` function with one parameter. The first uses a parameter of type `int` to set a gray value, the second uses a parameter of type `float` to set a gray value, and the third uses a parameter of type `color` to set a color value. The Processing language would be frustrating if a separate function name were used for each kind of fill.

This example uses three different `drawX()` functions, but all with the same name. The software knows which function to run by matching the number and type of the parameters.

21-15

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(255);  // Run first drawX()
  drawX(5.5);  // Run second drawX()
  drawX(0, 2, 44, 48, 36);  // Run third drawX()
}
```

21-15 cont.

```
// Draw an X with the gray value set by the parameter
void drawX(int gray) {
  stroke(gray);
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}

// Draw a black X with the thickness set by the parameter
void drawX(float weight) {
  stroke(0);
  strokeWeight(weight);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}

// Draws an X with the gray value, thickness,
// position, and size set by the parameters
void drawX(int gray, int weight, int x, int y, int s) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+s, y+s);
  line(x+s, y, x, y+s);
}
```

Calculating and returning values

In the examples shown so far, the output of a function has been shapes drawn to the screen. However, sometimes the preferred output is a number or other data. Data output from a function is called the return value. All functions are expected to return a value, such as an `int` or a `float`. If the function does not return a value, the special type `void` is used. The type of data returned by a function is found at the left of the function name.

The keyword `return` is used to exit a function and return to the location from which it was called. When a function outputs a value, `return` is used to specify what value should be returned. The `return` statement is typically the last line of a function because functions exit immediately after a `return`. We've already been using functions that return values. For example, `random()` returns a `float`, and the `color()` function returns a value of the `color` data type.

If a function returns a value, the function almost always appears to the right of an assignment operator or as a part of a larger expression. A function that does not return a value is often used as a complete statement. In the following example, notice how the value returned from the `random()` function is assigned to a variable, but the

ellipse() function is not associated with a variable. If the random() function is not assigned to a variable, the value will be lost .

```
float d = random(0, 100);
ellipse(50, 50, d, d);
```
21-16

When using functions that return values, it's important to be aware of the data type that is returned by each function. For example, the random() function returns floating-point values. If the result of the random() function is assigned to an integer, an error will occur.

```
int d = random(0, 100); // ERROR! random() returns floats
ellipse(50, 50, d, d);
```
21-17

The data-type conversion functions (p. 105) are useful for converting data into the format needed within a program. The previous example can be modified with the int() conversion function to match the type of data returned from random() to the type of data the result is assigned to:

```
int d = int(random(0, 100)); // int() converts the float value
ellipse(50, 50, d, d);
```
21-18

Consult the reference for each function to learn what data type is returned. Functions are not limited to returning numbers: they can return a PImage, String, boolean, or any other data type.

To write your own functions that return values, replace void with the data type you want to return. Include the return keyword inside your function to set the data to output. The value of the expression following the return will be output from the function. The following examples make useful calculations and return values, so they can be used elsewhere in the program.

```
void setup() {
  size(100, 100);
  float f = average(12.0, 6.0);  // Assign 9.0 to f
  println(f);
}

float average(float num1, float num2) {
  float av = (num1 + num2) / 2.0;
  return av;
}
```
21-19

21-20

```
void setup() {
  size(100, 100);
  float c = fahrenheitToCelsius(451.0); // Assign 232.77779 to c
  println(c);
}

float fahrenheitToCelsius(float t) {
  float f = (t-32.0) * (5.0/9.0);
  return f;
}
```

It's also important to note that you can't overload the return value of a function. Unlike functions that behave differently when given `float` or `int` values, it's not possible to have two functions with the same name that differ only in the type of data they return.

Exercises

1. *Write a function to draw a shape to the screen multiple times, each at a different position.*
2. *Extend the function created for exercise 1 by creating more parameters to control additional aspects of its form.*
3. *Write a function to use with a `for` structure to create a pattern evoking a liquid substance.*

Shape 3: Parameters, Recursion

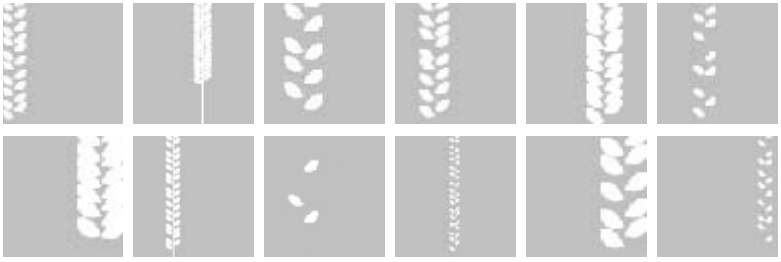
This unit introduces the concept of parameterized and recursive form.

Software provides a medium for exploring form in a unique way, and writing custom functions (p.181) enables such exploration. Using functions to generate shapes that vary based on their parameters is called parameterized form. A function can also contain a line of code that uses that same function—a technique called *recursion* that can be used in many ways to produce form.

Parameterized form

The leaf shape introduced in code 21-13 (p. 191) is an example of parameterized form. Different parameters passed into the `leaf()` function generate different forms:

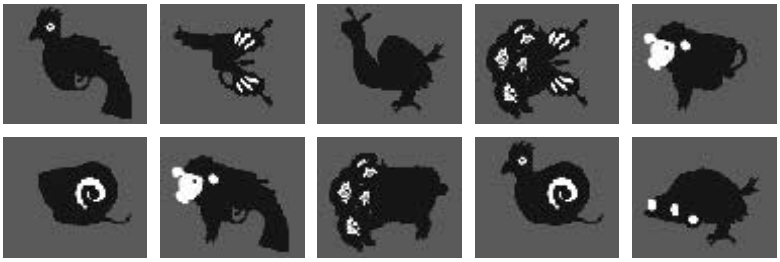
Parameterized form can grow in complexity when several functions are combined. This combination of functions allows one parameter to produce more diversity. Code 21-14 (p. 192) defines how the `vine()` function places a series of leaves. Changing the parameters to `vine()` produces a wide variety of shapes:



In the examples that use `leaf()`, the shape of the element remains the same, but the size and orientation changes. In the examples that use `vine()`, the size and quantity of elements change, but the leaf shapes remain constant. Parameterized form can also be used to change the shape of an element.

The clever Ars Magna cards created by Tatsuya Saito are an example of a modular image system. Nine images are each split into two cards:

Any front card can be used with any back card to produce unexpected results:



A program for producing random combinations of the cards follows.

22-01

```
size(120, 100);
int front = int(random(1, 10));  // Select the front card
int back = int(random(1, 10));   // Select the back card
PImage imgFront = loadImage(front + "f.jpg");
PImage imgBack = loadImage(back  + "b.jpg");
image(imgFront, 0, 0);
image(imgBack, 60, 0);
```

The Ars Magna system can create many unexpected image juxtapositions, but it offers only a finite number of possible options. Another way to create parameterized form uses the values input to a function to create continuous changes in the shape of a visual element. This is one of the greatest advantages of creating visual form with code.

A simple `arch()` function created using `bezierVertex()`, for example, can be continuously modulated by changes to a single parameter value. The parameter for `arch()` is a floating-point number, so it can be varied at extremely small measurements. A change in the parameter from 25.0001 to 25.0002 won't look different on screen, but will define a slightly different shape. The differences in shapes formed by using larger increments imply the possible shapes that lie between:

c=15.0

c=25.0

c=35.0

c=45.0

c=55.0

22-02

```
float c = 25.0;

void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  arch(c);
}

void arch(float curvature) {
  float y = 90.0;
  strokeWeight(6);
  noFill();
  beginShape();
  vertex(15.0, y);
  bezierVertex(15.0, y-curvature, 30.0, 55.0, 50.0, 55.0);
  bezierVertex(70.0, 55.0, 85.0, y-curvature, 85.0, y);
  endShape();
}
```

Within a parameterized system such as the `arch()` function, the value of one variable can affect the value of others. This is called coupling. If we change the code inside of the `arch()` function, the input parameter `curvature` can control the stroke thickness as well as the curvature.

c=15.0

c=35.0

c=55.0

22-03

```
void arch(float curvature) {
  float y = 90.0;
  float sw = (65.0 - curvature) / 4.0;
  strokeWeight(sw);
  noFill();
  beginShape();
  vertex(15.0, y);
  bezierVertex(15.0, y-curvature, 30.0, 55.0, 50.0, 55.0);
  bezierVertex(70.0, 55.0, 85.0, y-curvature, 85.0, y);
  endShape();
}
```

This is a modest example. The single input into `arch()` could be used to change every aspect of its display, including its values, rotation, size, etc. The following programs present more ideas related to the concept of parameterized form.

22-04

x=20
u=14
a=-0.12

x=40
u=9
a=-0.08

x=110
u=20
a=0.13

x=119
u=25
a=0.18

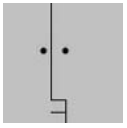
x=9
u=17
a=-0.18

```
int x = 20;       // X-coordinate
int u = 14;       // Units
float a = -0.12;  // Angle

void setup() {
  size(100, 100);
  stroke(0, 153);
  smooth();
  noLoop();
}

void draw() {
  background(204);
  tail(x, u, a);
}

void tail(int xpos, int units, float angle) {
  pushMatrix();
  translate(xpos, 0);
  for (int i = units; i > 0; i--) {  // Count in reverse
    strokeWeight(i);
    line(0, 0, 0, 8);
    translate(0, 8);
    rotate(angle);
  }
  popMatrix();
}
```

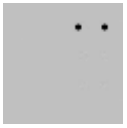
22-05

x=20
y=80
g=26

x=40
y=80
g=12



x=70
y=40
g=15



```
int x = 40;  // X-coordinate
int y = 30;  // Y-coordinate
int g = 20;  // Gap between eyes

void setup() {
  size(100, 100);
  smooth();
  fill(0);
  noLoop();
}

void draw() {
  background(204);
  face(x, y, g);
}
```

22-05 cont.

```
void face(int x, int y, int gap) {
  line(x, 0, x, y);                    // Nose Bridge
  line(x, y, x+gap, y);                // Nose
  line(x+gap, y, x+gap, 100);
  int mouthY = height - (height-y)/2;
  line(x, mouthY, x+gap, mouthY);      // Mouth
  noStroke();
  ellipse(x-gap/2, y/2, 5, 5);         // Left eye
  ellipse(x+gap, y/2, 5, 5);           // Right eye
}
```

Recursion

A common example of recursion is standing between two mirrors to see infinite reflections. In software, recursion means that a function can call itself within its own block. To prevent this from continuing forever, it's necessary to have some way for the function to exit. The following two programs produce the same result by different means, the first using a `for` structure and the second using recursion.

22-06

```
int x = 5;
for (int num = 15; num >= 0; num -= 1) {
  line(x, 20, x, 80);
  x += 5;
}
```

22-07

```
void setup() {
  drawLines(5, 15);
}

void drawLines(int x, int num) {
  line(x, 20, x, 80);
  if (num > 0) {
    drawLines(x+5, num-1);
  }
}
```

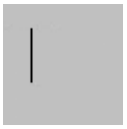
The recursive example uses more of the computer's resources to complete the task. For such a simple calculation, using the `for` structure is advised, but the recursive approach opens other possibilities. The following two examples utilize the custom `drawT()` function to show the effects of recursion.

x=50
y=100
a=35

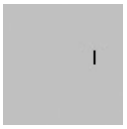
22-08

```
int x = 50;   // X-coordinate of the center
int y = 100;  // Y-coordinate of the bottom
int a = 35;   // Half the width of the top bar

void setup() {
  size(100, 100);
  noLoop();
}

void draw() {
  drawT(x, y, a);
}

void drawT(int xpos, int ypos, int apex) {
  line(xpos, ypos, xpos, ypos-apex);
  line(xpos-(apex/2), ypos-apex, xpos+(apex/2), ypos-apex);
}
```

x=24
y=65
a=45

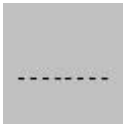


x=76
y=50
a=12



The `drawT()` function is made recursive by the inclusion of a call to itself within the function block. A fourth parameter called `num` is added to set the number of recursions. This value is decremented by 1 each time the function calls itself. When the value is no longer greater than 0, the recursion stops and the image is drawn to the screen.

x=50
a=20
n=8



22-09

```
int x = 50;   // X-coordinate of the center
int y = 100;  // Y-coordinate of the bottom
int a = 35;   // Half the width of the top bar
int n = 3;    // Number of recursions

void setup() {
  size(100, 100);
  noLoop();
}

void draw() {
  drawT(x, y, a, n);
}

void drawT(int x, int y, int apex, int num) {
  line(x, y, x, y-apex);
  line(x-apex, y-apex, x+apex, y-apex);
  // This relational expression must eventually be
  // false to stop the recursion and draw the lines
  if (num > 0) {
    drawT(x-apex, y-apex, apex/2, num-1);
```

x=50
a=35
n=3

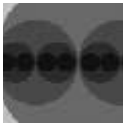
x=50
a=45
n=12

```
    drawT(x+apex, y-apex, apex/2, num-1);
  }
}
```

22-09 cont.

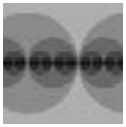
A binary tree structure (one that has two branches from each node) like the one above can be visualized in different ways. This program draws a circle at every node. The y-coordinate for each node is the same, and the radius for each circle is halved at each layer.

x=63
r=70
n=4



x=63
r=100
n=8

x=63
r=85
n=6

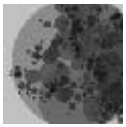


22-10

```
int x = 63;  // X-coordinate
int r = 85;  // Starting radius
int n = 6;   // Number of recursions

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  drawCircle(63, 85, 6);
}

void drawCircle(int x, int radius, int num) {
  float tt = 126 * num/4.0;
  fill(tt);
  ellipse(x, 50, radius*2, radius*2);
  if (num > 1) {
    num = num - 1;
    drawCircle(x - radius/2, radius/2, num);
    drawCircle(x + radius/2, radius/2, num);
  }
}
```

A slight modification yields a radical alteration of the form. Circles in every subsequent layer are given random positions relative to the previous position. The resulting images have a balance between order and disorder. At each level of recursion, the size of the circles decrease, their distance from the previous level decreases, and their values grow darker. Change the number used as the parameter to `randomSeed()` (p. 129) to produce a different composition.

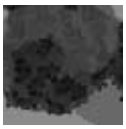
r=55 n=6 rs=18



r=65 n=6 rs=22

r=65 n=7 rs=22

r=90 n=6 rs=24



r=80 n=7 rs=12

r=90 n=6 rs=26

```
int x = 63;   // X-coordinate
int y = 50;   // Y-coordinate
int r = 80;   // Starting radius
int n = 7;    // Number of recursions
int rs = 12;  // Random seed value

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
  randomSeed(rs);
}

void draw() {
  drawCircle(x, y, r, n);
}

void drawCircle(float x, float y, int radius, int num) {
  float value = 126 * num / 6.0;
  fill(value, 153);
  ellipse(x, y, radius*2, radius*2);
  if (num > 1) {
    num = num - 1;
    int branches = int(random(2, 6));
    for (int i = 0; i < branches; i++) {
      float a = random(0, TWO_PI);
      float newx = x + cos(a) * 6.0 * num;
      float newy = y + sin(a) * 6.0 * num;
      drawCircle(newx, newy, radius/2, num);
    }
  }
}
```

Exercises

1. *Write your own function to draw a parameterized arch.*
2. *Create a function for drawing a chair. Use two parameters to change its position and two more to change the shape. Using your function, draw 9 chairs in the display window in a regular 3 × 3 matrix. Use different parameters to give each chair drawn a unique shape.*
3. *Modify code 22-04 to create a sequence of different compositions.*

Input 1: Mouse I

This unit introduces mouse input as a way to control the position and attributes of shapes on screen. It also explains how to change the cursor icon.

Syntax introduced:
`mouseX, mouseY, pmouseX, pmouseY, mousePressed, mouseButton`
`cursor(), noCursor()`

The screen forms a bridge between our bodies and the realm of circuits and electricity inside computers. We control elements on screen through a variety of devices such as touch pads, trackballs, and joysticks, but—aside from the keyboard—the most common input device is the mouse. The computer mouse dates back to the late 1960s when Douglas Engelbart presented the device as an element of the oN-Line System (NLS), one of the first computer systems with a video display. The mouse concept was further developed at the Xerox Palo Alto Research Center (PARC), but its introduction with the Apple Macintosh in 1984 was the catalyst for its current ubiquity. The design of the mouse has gone through many revisions in the last thirty years, but its function has remained the same. In Engelbart's original patent application in 1970 he referred to the mouse as an "X-Y position indicator," and this still accurately, but dryly, defines its contemporary use.

The physical mouse object is used to control the position of the cursor on screen and to select interface elements. The cursor position is read by computer programs as two numbers, the x-coordinate and the y-coordinate. These numbers can be used to control attributes of elements on screen. If these coordinates are collected and analyzed, they can be used to extract higher-level information such as the speed and direction of the mouse. This data can in turn be used for gesture and pattern recognition.

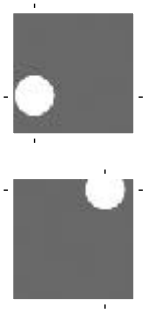
Mouse data

The Processing variables `mouseX` and `mouseY` (note the capital X and Y) store the x-coordinate and y-coordinate of the cursor relative to the origin in the upper-left corner of the display window. To see the actual values produced while moving the mouse, run this program to print the values to the console:

23-01
```
void draw() {
  frameRate(12);
  println(mouseX + " : " + mouseY);
}
```

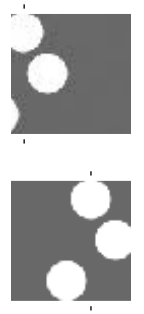
When a program starts, `mouseX` and `mouseY` values are 0. If the cursor moves into the display window, the values are set to the current position of the cursor. If the cursor is at the left, the `mouseX` value is 0 and the value increases as the cursor moves to the right. If the cursor is at the top, the `mouseY` value is 0 and the value increases as the cursor moves down. If `mouseX` and `mouseY` are used in programs without a `draw()` or if `noLoop()` is run in `setup()`, the values will always be 0.

The mouse position is most commonly used to control the location of visual elements on screen. More interesting relations are created when the visual elements relate differently to the mouse values, rather than simply mimicking the current position. Adding and subtracting values from the mouse position creates relationships that remain constant, while multiplying and dividing these values creates changing visual relationships between the mouse position and the elements on the screen. To invert the value of the mouse, simply subtract the `mouseX` value from the width of the window and subtract the `mouseY` value from the height of the screen.



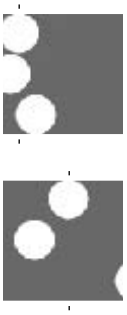
23-02

```
// Circle follows the cursor (the cursor position is
// implied by the crosshairs around the illustration)

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(126);
  ellipse(mouseX, mouseY, 33, 33);
}
```



23-03

```
// Add and subtract to create offsets

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(126);
  ellipse(mouseX, 16, 33, 33);     // Top circle
  ellipse(mouseX+20, 50, 33, 33);  // Middle circle
  ellipse(mouseX-20, 84, 33, 33);  // Bottom circle
}
```

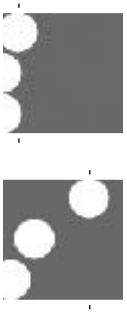


23-04

```
// Multiply and divide to creates scaling offsets

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(126);
  ellipse(mouseX, 16, 33, 33);    // Top circle
  ellipse(mouseX/2, 50, 33, 33);  // Middle circle
  ellipse(mouseX*2, 84, 33, 33);  // Bottom circle
}
```

23-05

```
// Invert cursor position to create a secondary response

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  float x = mouseX;
  float y = mouseY;
  float ix = width - mouseX;   // Inverse X
  float iy = mouseY - height;  // Inverse Y
  background(126);
  fill(255, 150);
  ellipse(x, height/2, y, y);
  fill(0, 159);
  ellipse(ix, height/2, iy, iy);
}
```

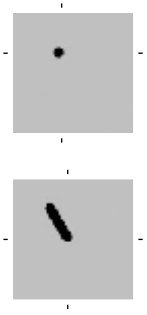


```
// Exponential functions can create nonlinear relations      23-06
// between the mouse and shapes affected by the mouse

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(126);
  float normX = mouseX / float(width);
  ellipse(mouseX, 16, 33, 33);                  // Top
  ellipse(pow(normX, 4) * width, 50, 33, 33);  // Middle
  ellipse(pow(normX, 8) * width, 84, 33, 33);  // Bottom
}
```

The Processing variables `pmouseX` and `pmouseY` store the mouse values from the previous frame. If the mouse does not move, the values will be the same, but if the mouse is moving quickly there can be large differences between the values. To see the difference, run the following program and alternate moving the mouse slowly and quickly. Watch the values print to the console.

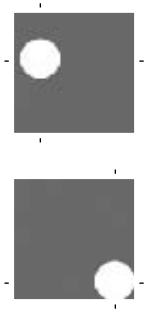
```
void draw() {                                                23-07
  frameRate(12);
  println(pmouseX - mouseX);
}
```

Drawing a line from the previous mouse position to the current position shows the changing position in one frame, revealing the speed and direction of the mouse. When the mouse is not moving, a point is drawn, but quick mouse movements create long lines.



```
// Draw a line between the current and previous positions    23-08

void setup() {
  size(100, 100);
  strokeWeight(8);
  smooth();
}

void draw() {
  background(204);
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

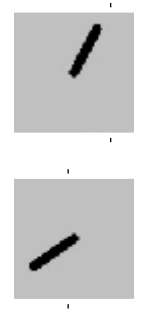
The mouseX and mouseY values can control translation, rotation, and scale by using them as parameters in the transformation functions. You can move a circle around the screen by changing the parameters to translate() rather than by changing the *x* and *y* parameters of ellipse().



23-09

```
// Use translate() to move a shape

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(126);
  translate(mouseX, mouseY);
  ellipse(0, 0, 33, 33);
}
```

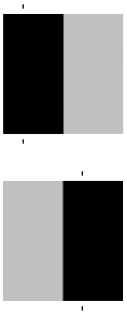
Before using mouseX and mouseY as parameters to transformation functions, it's important to think first about how they relate to the expected parameters. For example, the rotate() function expects its parameters in units of radians (p. 117). To make a shape rotate 360 degrees as the cursor moves from the left edge to the right edge of the window, the values of mouseX must be converted to values from 0.0 to 2π. In the following example, the map() function is used to make this conversion. The resulting value is used as the parameter to rotate() to turn the line as the mouse moves back and forth between the left and right edge of the display window.



23-10

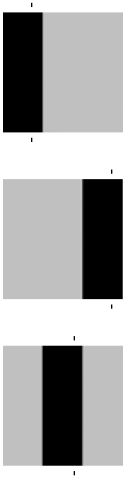
```
// Use rotate() to move a shape

void setup() {
  size(100, 100);
  strokeWeight(8);
  smooth();
}

void draw() {
  background(204);
  float angle = map(mouseX, 0, width, 0, TWO_PI);
  translate(50, 50);
  rotate(angle);
  line(0, 0, 40, 0);
}
```

Using the `mouseX` and `mouseY` variables with an `if` structure allows the cursor to select regions of the screen. The following examples demonstrate the cursor making a selection between different areas of the display window.



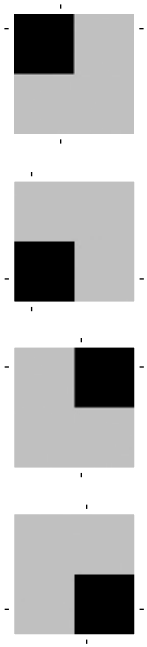
23-11

```
// Cursor position selects the left or right half
// of the display window

void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}

void draw() {
  background(204);
  if (mouseX < 50) {
    rect(0, 0, 50, 100);   // Left
  } else {
    rect(50, 0, 50, 100);  // Right
  }
}
```



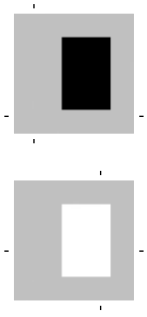
23-12

```
// Cursor position selects the left, middle,
// or right third of the display window

void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}

void draw() {
  background(204);
  if (mouseX < 33) {
    rect(0, 0, 33, 100);   // Left
  } else if ((mouseX >= 33) && (mouseX <= 66)) {
    rect(33, 0, 33, 100);  // Middle
  } else {
    rect(66, 0, 33, 100);  // Right
  }
}
```



23-13

```
// Cursor position selects a quadrant of
// the display window

void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}

void draw() {
  background(204);
  if ((mouseX <= 50) && (mouseY <= 50)) {
    rect(0, 0, 50, 50);    // Upper-left
  } else if ((mouseX <= 50) && (mouseY > 50)) {
    rect(0, 50, 50, 50);   // Lower-left
  } else if ((mouseX > 50) && (mouseY < 50)) {
    rect(50, 0, 50, 50);   // Upper-right
  } else {
    rect(50, 50, 50, 50);  // Lower-right
  }
}
```

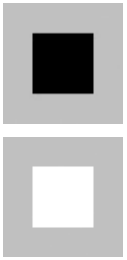


23-14

```
// Cursor position selects a rectangular area to
// change the fill color

void setup() {
  size(100, 100);
  noStroke();
  fill(0);
}

void draw() {
  background(204);
  if ((mouseX > 40) && (mouseX < 80) &&
      (mouseY > 20) && (mouseY < 80)) {
    fill(255);
  } else {
    fill(0);
  }
  rect(40, 20, 40, 60);
}
```

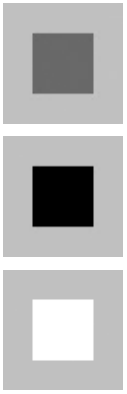
Mouse buttons

Computer mice and other similar input devices typically have between one and three buttons, and Processing can detect when these buttons are pressed. The button status and the cursor position together allow the mouse to perform different actions. For example, pressing a button when the mouse is over an icon can select it, so the icon can be moved to a different location on screen. The `mousePressed` variable is `true` if any mouse button is pressed and `false` if no mouse button is pressed. The variable `mouseButton` is `LEFT`, `CENTER`, or `RIGHT` depending on the mouse button most recently pressed. The `mousePressed` variable reverts to `false` as soon as the button is released, but the `mouseButton` variable retains its value until a different button is pressed. These variables can be used independently or in combination to control your software. Run these programs to see how the software responds to your fingers.



23-15

```
// Set the square to white when a mouse button is pressed

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  if (mousePressed == true) {
    fill(255);  // White
  } else {
    fill(0);    // Black
  }
  rect(25, 25, 50, 50);
}
```

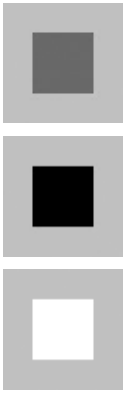


23-16

```
// Set the square to black when the left mouse button
// is pressed and white when the right button is pressed

void setup() {
  size(100, 100);
}

void draw() {
  if (mouseButton == LEFT) {
    fill(0);    // Black
  } else if (mouseButton == RIGHT) {
    fill(255);  // White
  } else {
```

```
    fill(126);  // Gray
  }
  rect(25, 25, 50, 50);
}
```

23-16 cont.



```
// Set the square to black when the left mouse button
// is pressed, white when the right button is pressed,
// and gray when a button is not pressed

void setup() {
  size(100, 100);
}

void draw() {
  if (mousePressed == true) {
    if (mouseButton == LEFT) {
      fill(0);    // Black
    } else if (mouseButton == RIGHT) {
      fill(255);  // White
    }
  } else {
    fill(126);    // Gray
  }
  rect(25, 25, 50, 50);
}
```

23-17

Not all mice have multiple buttons, and if software is distributed widely, the interaction should not rely on detecting which button is pressed. For example, if you are posting your work on the Web, don't rely on the middle or right button for using the software because many users won't have a two- or three-button mouse.

Cursor icon

The cursor can be hidden with the `noCursor()` function and can be set to appear as a different icon with the `cursor()` function. When the `noCursor()` function is run, the cursor icon disappears as it moves into the display window. To give feedback about the location of the cursor within the software, a custom cursor can be drawn and controlled with the `mouseX` and `mouseY` variables.

```
// Draw an ellipse to show the position of the hidden cursor                23-18

void setup() {
  size(100, 100);
  strokeWeight(7);
  smooth();
  noCursor();
}

void draw() {
  background(204);
  ellipse(mouseX, mouseY, 10, 10);
}
```

If `noCursor()` is run, the cursor will be hidden while the program is running until the `cursor()` function is run to reveal it.

```
// Hides the cursor until a mouse button is pressed                         23-19

void setup() {
  size(100, 100);
  noCursor();
}

void draw() {
  background(204);
  if (mousePressed == true) {
    cursor();
  }
}
```

Adding a parameter to the `cursor()` function allows it to be changed to another icon. The self-descriptive options for the *MODE* parameter are ARROW, CROSS, HAND, MOVE, TEXT, and WAIT.

```
// Draws the cursor as a hand when a mouse button is pressed                23-20

void setup() {
  size(100, 100);
  smooth();
}

void draw() {
  background(204);
```

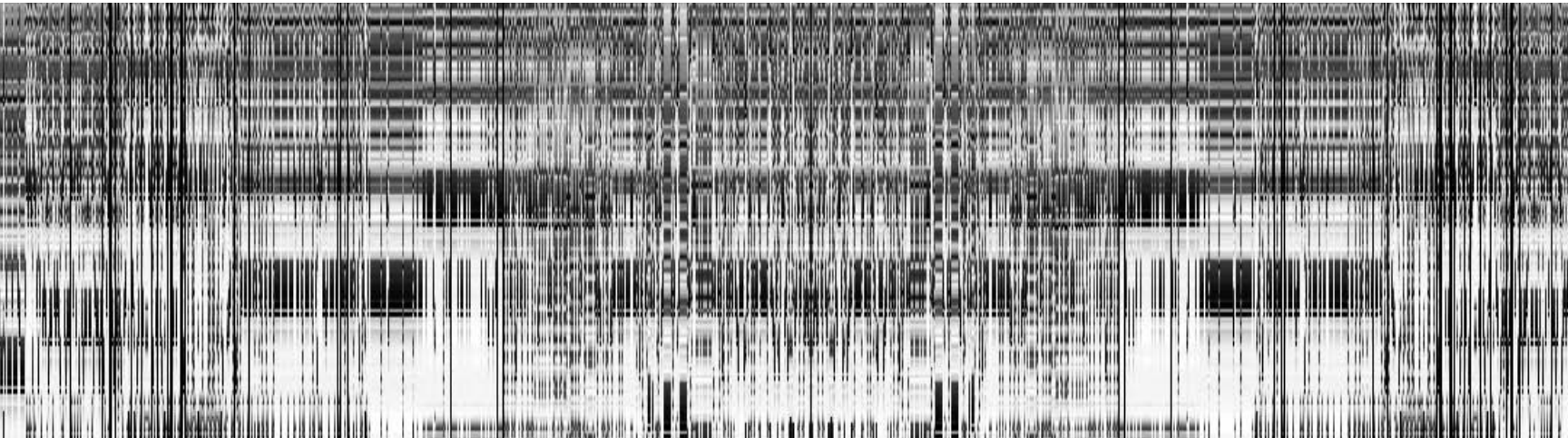
```
  if (mousePressed == true) {
    cursor(HAND);
  } else {
    cursor(MOVE);
  }
  line(mouseX, 0, mouseX, height);
  line(0, mouseY, height, mouseY);
}
```

23-20
cont.

These cursor images are part of your computer's operating system and will appear differently on different machines.

Exercises

1. *Control the position of a shape with the mouse. Strive to create a more interesting relation than one directly mimicking the position of the cursor.*
2. *Invent three unique shapes that behave differently in relation to the mouse. Each shape's behavior should change when the mouse is pressed. Relate the form of each shape to its behavior.*
3. *Create a custom cursor that changes as the mouse moves through the display window.*



Drawing 1: Static Forms

This unit discusses drawing in relation to software and presents code for basic drawing programs.

The activity of drawing translates an individual's perception and imagination into visual form. The differences between the drawings of different people demonstrates the fact that every hand and mind is unique. Drawings range from the mechanical grids of Sol LeWitt to the playful lines of Paul Klee to the expressionist figures of Egon Schiele and far beyond. Each surface and instrument offers a different tactile experience. Ink, charcoal, crayon, vellum, and cloth all enable unique sorts of drawing. Less conventional materials, such as chocolate, dirt, glue, and light, have been used with significant results. Materials can be applied with fingers, toes, or elbows or with utensils like pencils, brushes, and sticks. Every choice of material and application influences the viewer's perception of what the composition communicates.

The idea of drawing with computers dates back to the 1960s. Ivan Sutherland created the remarkable Sketchpad software for his PhD dissertation in 1963. Sketchpad, the progenitor of computer-aided drawing software (CAD) such as Autodesk's AutoCAD and Adobe Illustrator, used the newly invented light pen input device that made it possible to draw directly to the screen. The software had features to convert imprecise marks into perfect straight lines, arcs, and circles. It could also constrain marks to make them identical, parallel, or perpendicular. Most example drawings demonstrated the features and accuracy of the system for making technical drawings, but Sutherland also discussed the use of his software for other purposes and created an example of an animated portrait.

Logo is another innovative software drawing system with origins in the 1960s. Seymour Papert developed Logo's turtle graphics as a way to get children thinking about geometry. Logo uses text commands to control a turtle on the screen that leaves a trail as it travels. The command `RT 90` turns the turtle 90° to the right, and `FD 100` moves the turtle forward 100 units. One early Logo implementation employed a robotic turtle named Irving that moved around the room according to the children's instructions.

In contrast to the interactive approach of Sketchpad and Logo, most early software drawing systems translated input directly from code to paper. Software drawing pioneers of the early 1960s included A. Michael Noll, Frieder Nake, Georg Nees, and Charles Csuri. Their images were realized with computer-driven plotters, a common output device of that time. A plotter is a pen attached to a moving mechanical arm controlled by motors through a computer. Many drawings from this period utilized the technology in a way that showcased the precision of the tools. Another wave of individuals utilizing software plotters emerged in the 1970s and 1980s. These artists included Manfred Mohr, Jean-Pierre Hébert, and Mark Wilson.

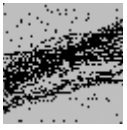
Bridging the era of plotters to present-day technologies, Harold Cohen's *AARON* is arguably the most sophisticated drawing software ever written. The software has undergone continuous development since it was first created in 1973. *AARON*'s drawings have been featured in some of the world's most prominent museums, including the Tate Gallery in London and the Stedelijk Museum in Amsterdam. *AARON* initially created abstract drawings and over the years has been refined to add rocks, then plants, and finally people. Cohen has encoded his ideas about drawing as a set of rules that comprise the *AARON* software. The program operates autonomously and makes a unique drawing each time it is run. The software makes every composition decision and produces drawings that often surprise Cohen.

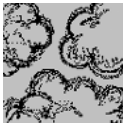
Contemporary artists continue to write innovative software to enable unique approaches to drawing. Input tools can severely limit drawing with software. The quality of drawing with a device such as the mouse is constrained by the small amount of information transferred from the hand into the software. The hand, with its strength and flexibility, is capable of gestures of the smallest nuance in pressure and direction, but the mouse receives only position information. Such limitations were diminished with the introduction of drawing tablets and stylus devices capable of reading pressure and direction, but no matter how much these devices improve, they will only approximate the quality of using physical instruments and media.

Rather than applying a "traditional" model of drawing to the software medium, another approach is to address the possibilities tangential to the constraints. When using a new medium, why constrain oneself to imitating other media? In the context of this book, we consider the potential of software in contrast to physical media and address drawing methods that are unique to software.

Simple tools

The easiest way to draw with Processing is to not include the `background()` function inside `draw()`. This omission allows the display window to accumulate pixels from frame to frame.

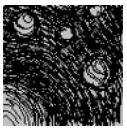




24-01

```
// Draw dots at the position of the cursor

void setup() {
  size(100, 100);
}

void draw() {
  point(mouseX, mouseY);
}
```

```
// Draw from the previous mouse location to the current          24-02
// mouse location to create a continuous line

void setup() {
  size(100, 100);
}

void draw() {
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```



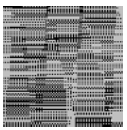
```
// Draw a line only when a mouse button is pressed               24-03

void setup() {
  size(100, 100);
}

void draw() {
  if (mousePressed == true) {
    line(mouseX, mouseY, pmouseX, pmouseY);
  }
}
```

```
// Draw lines with different gray values when a mouse            24-04
// button is pressed or not pressed

void setup() {
  size(100, 100);
}

void draw() {
  if (mousePressed == true) {  // If mouse is pressed,
    stroke(255);               // set the stroke to white
  } else {                     // Otherwise,
    stroke(0);                 // set to black
  }
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Drawing with software is not restricted to making a single mark that follows the cursor. A `for` structure makes it possible to create more complex drawings with just a few lines of code. The following examples use a `for` structure to draw many elements to the screen at each frame.

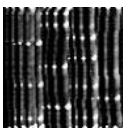
24-05

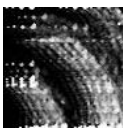
```
void setup() {
  size(100, 100);
}

void draw() {
  for (int i = 0; i < 50; i += 2) {
    point(mouseX+i, mouseY+i);
  }
}
```



24-06

```
void setup() {
  size(100, 100);
}

void draw() {
  for (int i = -14; i <= 14; i += 2) {
    point(mouseX+i, mouseY);
  }
}
```





24-07

```
void setup() {
  size(100, 100);
  noStroke();
  fill(255, 40);
  background(0);
}

void draw() {
  if (mousePressed == true) {
    fill(0, 26);
  } else {
    fill(255, 26);
  }
  for (int i = 0; i < 6; i++) {
    ellipse(mouseX + i*i, mouseY, i, i);
  }
}
```

Drawing with images

Images can also be used as drawing tools. If an image is positioned in relation to the cursor at each frame, its pixels can be used to create visually complex compositions. In the following examples, the image follows the position of the cursor.

24-08

```
// Draw with an image sliver

PImage lineImage;

void setup() {
  size(100, 100);
  // This image is 100 pixels wide, but one pixel tall
  lineImage = loadImage("imageline.jpg");
}

void draw() {
  image(lineImage, mouseX-lineImage.width/2, mouseY);
}
```

24-09

```
// Draw with an image that has transparency

PImage alphaImg;

void setup() {
  size(100, 100);
  // This image is partially transparent
  alphaImg = loadImage("alphaArch.png");
}

void draw() {
  int ix = mouseX - alphaImg.width/2;
  int iy = mouseY - alphaImg.height/2;
  image(alphaImg, ix, iy);
}
```

Exercises

1. *Make a custom drawing tool that changes its color when a mouse button is pressed.*
2. *Make a custom drawing tool that changes its form when a mouse button is pressed.*
3. *Load an image and use it as a drawing tool.*

1 | 2 | 3 | 4 | 5 | 6

Q | W | E | R | T | Y

A | S | D | F | G |

Z | X | C | V | B |

Input 2: Keyboard

This unit introduces keyboard input.

Syntax introduced:
`keyPressed, key, keyCode`

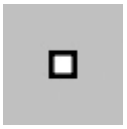
Keyboards are typically used to input characters for composing documents, Email, and instant messages, but the keyboard has potential for use beyond its original intent. The migration of the keyboard from typewriter to computer expanded its function to enable launching software, moving through the menus of software applications, and navigating 3D environments in games. When writing your own software, you have the freedom to use the keyboard data any way you wish. For example, basic information such as the speed and rhythm of the fingers can be determined by the rate at which keys are pressed. This information could control the speed of an event or the quality of motion. It's also possible to ignore the characters printed on the keyboard itself and use the location of each key relative to the keyboard grid as a numeric position.

The modern computer keyboard is a direct descendant of the typewriter. The position of the keys on an English-language keyboard is inherited from early typewriters. This layout is called QWERTY because of the order of the top row of letter keys. It was developed for typewriters to put physical distance between frequently typed letter pairs, helping reduce the likelihood of the typebars colliding and jamming as they hit the ribbon. There are variations on this layout for different languages including the AWERTY layout for the French language and the QWERTZ layout for German. The alphabetic differences are small, but the symbol placement on these keyboard variations can be extreme. For example, commonly used programming symbols such as { and } are not printed on French keyboards, but can be accessed through the Alt Gr key. Keyboards for languages with different alphabets often keep the same physical key arrangement but replace the characters with, for example, Greek, Arabic, or Thai characters. Keyboards for alphabets with thousands of characters, such as Chinese, don't have a direct mapping between key and character—they use a system where a series of key presses is interpreted by the operating system to build each symbol.

In recent years we've seen the dominance of keyboards challenged by alternate input methods on small, handheld devices such as the Palm Pilot. Although the Palm's Graffiti software makes it simple to translate hand gestures into characters, the mini-keyboards on mobile phones and other personal digital assistants have reconfirmed many people's preference for inputting data with a keyboard. Some users have become adept at typing characters with a mobile-phone keypad, while others opt for phones with miniature QWERTY keyboards. On the other hand, speech recognition is always improving and provides an alternative to the keyboard for certain kinds of tasks.

Keyboard data

Processing registers the most recently pressed key and whether a key is currently pressed. The `boolean` variable `keyPressed` is `true` if a key is pressed and is `false` if not. Including this variable in an `if` structure allows lines of code to run only if a key is pressed. The `keyPressed` variable remains `true` while the key is held down and becomes `false` only when the key is released.



25-01

```
// Draw a rectangle while any key is pressed

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}

void draw() {
  background(204);
  if (keyPressed == true) {  // If the key is pressed,
    line(20, 20, 80, 80);    // draw a line
  } else {                   // Otherwise,
    rect(40, 40, 20, 20);    // draw a rectangle
  }
}
```

25-02

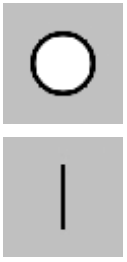
```
// Move a line while any key is pressed

int x = 20;

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}

void draw() {
  background(204);
  if (keyPressed == true) {  // If the key is pressed
    x++;                     // add 1 to x
  }
  line(x, 20, x-60, 80);
}
```

The `key` variable is of the `char` data type and stores the most recently pressed key. The `key` variable can store only one value at a time. The most recent key pressed will be the only value stored in the variable. A `key` can be displayed on screen by loading a font and using the `text()` function (p. 112).

25-03

```
PFont font;

void setup() {
  size(100, 100);
  font = loadFont("ThesisMonoLight-72.vlw");
  textFont(font);
}

void draw() {
  background(0);
  text(key, 28, 75);
}
```

The `key` variable may be used to determine whether a specific key is pressed. The following example uses the expression `key == 'A'` to test if the *A* key is pressed. The single quotes signify *A* as the data type `char`. The expression `key == "A"` will cause an error because the double quotes signify the *A* as a `String`, and it's not possible to compare a `String` with a `char`. The logical AND symbol, the && operator, is used to connect the expression with the `keyPressed` variable to ascertain that the key pressed is the uppercase *A*.



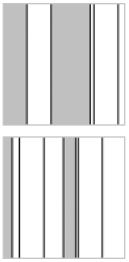
25-04

```
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}

void draw() {
  background(204);
  // If the 'A' key is pressed draw a line
  if ((keyPressed == true) && (key == 'A')) {
    line(50, 25, 50, 75);
  } else {  // Otherwise, draw an ellipse
    ellipse(50, 50, 50, 50);
  }
}
```

If you want to check for both uppercase and lowercase letters, you have to extend the relational expression with a logical OR, the || relational operator. Line 10 in the previous program would be changed to

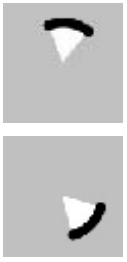
```
if ((keyPressed == true) && ((key == 'a') || (key == 'A'))) {
```

Because each character has a numeric value as defined by the ASCII table (p. 665), the value of the `key` variable can be used to control visual attributes such as the position and color of shape elements.



25-05

```
int x = 0;

void setup() {
  size(100, 100);
}

void draw() {
  if (keyPressed == true) {
    x = key - 32;
    rect(x, -1, 20, 101);
  }
}
```

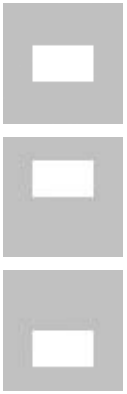


25-06

```
float angle = 0;

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(8);
}

void draw() {
  background(204);
  if (keyPressed == true) {
    if ((key >= 32) && (key <= 126)) {
      // If the key is alphanumeric,
      // convert its value into an angle
      angle = map(key, 32, 126, 0, TWO_PI);
    }
  }
  arc(50, 50, 66, 66, angle-PI/6, angle+PI/6);
}
```

Coded keys

In addition to reading key values for numbers, letters, and symbols, Processing can also read the values from other keys including the arrow keys and the Alt, Control, Shift, Backspace, Tab, Enter, Return, Escape, and Delete keys. The variable `keyCode` stores the `ALT`, `CONTROL`, `SHIFT`, `UP`, `DOWN`, `LEFT`, and `RIGHT` keys as constants. Before determining which coded key is pressed, it's necessary to check first to see if the key is coded. The expression `key == CODED` is `true` if the key is coded and `false` otherwise. Even though not alphanumeric, the keys included in the ASCII (p. 664) specification (`BACKSPACE`, `TAB`, `ENTER`, `RETURN`, `ESC`, and `DELETE`) will not be identified as a coded key. If you're making cross-platform projects, note that the Enter key is commonly used on PCs and UNIX and the Return key is used on Macintosh. Check for both Enter and Return to make sure your program will work for all platforms (see code 26-08).

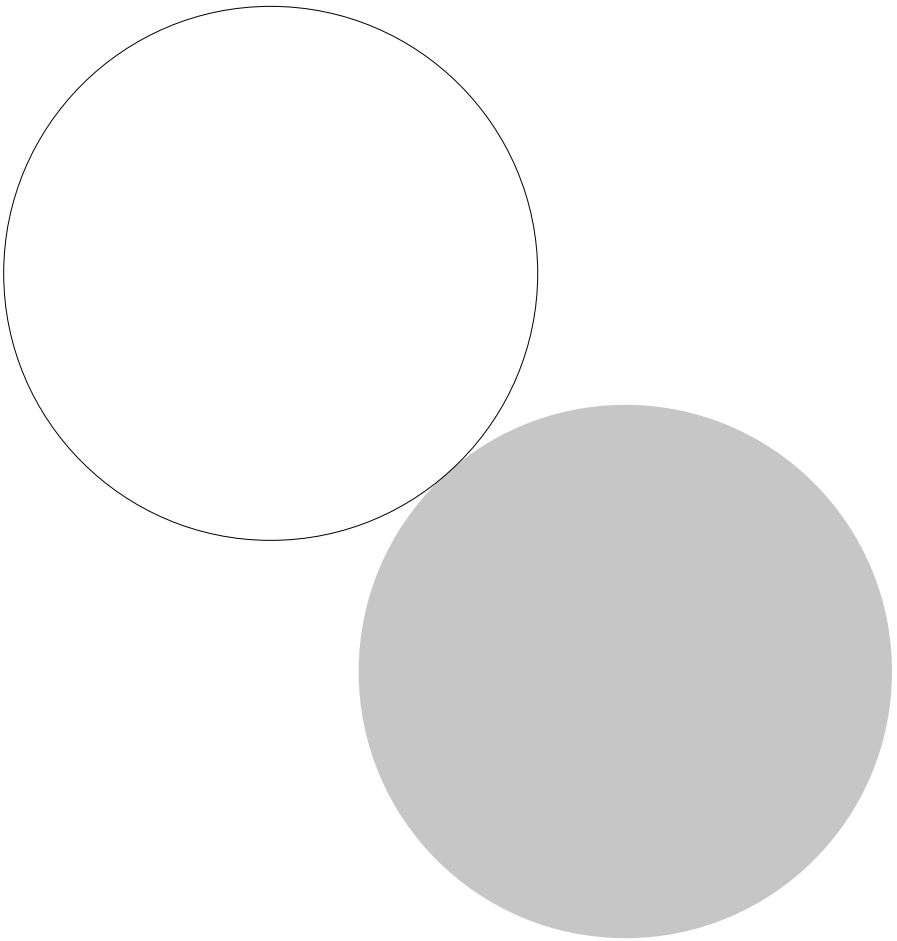


25-07

```
int y = 35;

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  line(10, 50, 90, 50);
  if (key == CODED) {
    if (keyCode == UP) {
      y = 20;
    } else if (keyCode == DOWN) {
      y = 50;
    }
  } else {
    y = 35;
  }
  rect(25, y, 50, 30);
}
```

Exercises

1. *Use the number keys on the keyboard to modify the movement of a line.*
2. *Create a typing program to display a different image for each letter on the keyboard.*
3. *Use the arrow keys to change the position of a shape within the display window.*



Input 3: Events

This unit introduces mouse and keyboard events for detecting actions and receiving data.

Syntax introduced:
`mousePressed(), mouseReleased(), mouseMoved(), mouseDragged()`
`keyPressed(), keyReleased()`
`loop(), redraw()`

Functions called events alter the normal flow of a program when an action such as a key press or mouse movement takes place. An event is a polite interruption of the normal flow of a program. Key presses and mouse movements are stored until the end of `draw()`, where they can take action that won't disturb drawing that's currently in progress. The code inside an event function is run once each time the corresponding event occurs. For example, if a mouse button is pressed, the code inside the `mousePressed()` function will run once and will not run again until the button has been released and is pressed again. This allows data produced by the mouse and keyboard to be read independently from what is happening in the rest of the program. It is also more accurate, because even if the mouse moves several times before `draw()` is finished, the value for `mouseX` and `mouseY` will be the same throughout the method (again, this avoids interruption of a composition in the middle of drawing).

Mouse events

The mouse event functions are `mousePressed()`, `mouseReleased()`, `mouseMoved()`, and `mouseDragged()`:

`mousePressed()`	*Code inside this block is run one time when a mouse button is pressed*
`mouseReleased()`	*Code inside this block is run one time when a mouse button is released*
`mouseMoved()`	*Code inside this block is run one time when the mouse is moved*
`mouseDragged()`	*Code inside this block is run one time when the mouse is moved while a mouse button is pressed*

The `mousePressed()` function works differently than the `mousePressed` variable discussed in Input 1 (p. 205). The value of the `mousePressed` variable is `true` until the mouse button is released. It can therefore be used within `draw()` to have a line of code run while the mouse is pressed. In contrast, the code inside the `mousePressed()` function only runs once when a button is pressed. This makes it useful when a mouse click is used to trigger an action, such as clearing the screen. In the following example, the background value becomes lighter each time a mouse button is pressed. Run the example on your computer to see the change in response to your finger.

```
float gray = 0;

void setup() {
  size(100, 100);
}

void draw() {
  background(gray);
}

void mousePressed() {
  gray += 20;
}
```

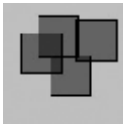
26-01

The following example is the same as the one above, but the `gray` variable is set in the `mouseReleased()` event function, which is called once every time a key is released. This difference can be seen only by running the program and clicking the mouse button. Keep the mouse button pressed for a long time and notice that the background value changes only when the button is released.

```
float gray = 0;

void setup() {
  size(100, 100);
}

void draw() {
  background(gray);
}

void mouseReleased() {
  gray += 20;
}
```

26-02

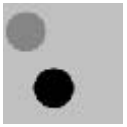
Before drawing inside these functions, it's important to think about the flow of the program. In this example, circles are drawn inside `mousePressed()` and they remain on screen because there is no `background()` inside `draw()`. But if `background()` is used, visual elements drawn within one of the mouse event functions will appear on screen for only a single frame. In fact, you'll notice this example has nothing at all inside `draw()`, but it needs to be there to force Processing to keep listening for the events. If a `background()` function were run inside `draw()`, the rectangles would flash onto the screen and disappear.

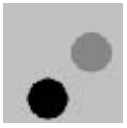


26-03

```
void setup() {
  size(100, 100);
  fill(0, 102);
}

void draw() { }  // Empty draw() keeps the program running

void mousePressed() {
  rect(mouseX, mouseY, 33, 33);
}
```

The code inside the `mouseMoved()` and `mouseDragged()` event functions is run when there is a change in the mouse position. The code in the `mouseMoved()` block is run at the end of each frame when the mouse moves and no button is pressed. The code in the `mouseDragged()` block does the same when the mouse button is pressed. If the mouse stays in the same position from frame to frame, the code inside these functions does not run. In this example, the gray circle follows the mouse when the button is not pressed and the white circle follows the mouse when a mouse button is pressed.





26-04

```
int dragX, dragY, moveX, moveY;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(204);
  fill(0);
  ellipse(dragX, dragY, 33, 33);  // Black circle
  fill(153);
  ellipse(moveX, moveY, 33, 33);  // Gray circle
}

void mouseMoved() {    // Move gray circle
  moveX = mouseX;
  moveY = mouseY;
}

void mouseDragged() {  // Move black circle
  dragX = mouseX;
  dragY = mouseY;
}
```

Key events

Each key press is registered through the keyboard event functions `keyPressed()` and `keyReleased()`:

`keyPressed()`	*Code inside this block is run one time when any key is pressed*
`keyReleased()`	*Code inside this block is run one time when any key is released*

Each time a key is pressed, the code inside the `keyPressed()` block is run once.[1] Within this block, it's possible to test which key has been pressed and to use this value for any purpose. In this example, the numeric value of the key is used to position a white rectangle on the screen.

26-05

```
void setup() {
  size(100, 100);
  noStroke();
  fill(255, 51);
}

void draw() { }  // Empty draw() keeps the program running

void keyPressed() {
  int y = key - 32;
  rect(0, y, 100, 4);
}
```

Each time a key is released, the code inside the `keyReleased()` block is run once. In the following example, each time the *T* key is pressed, a `boolean` variable is set to `true` that allows a T shape to display within `draw()`. When the key is released, this `boolean` is set to false and the shape is no longer displayed.

26-06

```
boolean drawT = false;

void setup() {
  size(100, 100);
  noStroke();
}

void draw() {
  background(204);
  if (drawT == true) {
    rect(20, 20, 60, 20);
    rect(39, 40, 22, 45);
  }
}
```

```
void keyPressed() {
  if ((key == 'T') || (key == 't')) {
    drawT = true;
  }
}

void keyReleased() {
  drawT = false;
}
```

26-06 cont.

The following two examples use `keyPressed()` to read and analyze input from the keyboard. Each utilizes the `String` methods introduced in Data 3 (p. 105).

26-07

```
// An extremely minimal text editor, it can only insert
// and remove characters from a single line

PFont font;
String letters = "";

void setup() {
  size(100, 100);
  font = loadFont("Eureka-24.vlw");
  textFont(font);
  stroke(255);
  fill(0);
}

void draw() {
  background(204);
  float cursorPosition = textWidth(letters);
  line(cursorPosition, 0, cursorPosition, 100);
  text(letters, 0, 50);
}

void keyPressed() {
  if (key == BACKSPACE) {  // Backspace
    if (letters.length() > 0) {
      letters = letters.substring(0, letters.length()-1);
    }
  } else if (textWidth(letters+key) < width){
    letters = letters+key;
  }
}
```

```
// Compare the input from the keyboard to see if it's
// either "black" or "gray" and set the background
// value accordingly. Press Enter or Return to input
// the data

PFont font;
String letters = "";
int back = 102;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-24.vlw");
  textFont(font);
  textAlign(CENTER);
}

void draw() {
  background(back);
  text(letters, 50, 50);
}

void keyPressed() {
  if ((key == ENTER) || (key == RETURN)) {
    letters = letters.toLowerCase();
    println(letters);  // Print to console to see input
    if (letters.equals("black")) {
      back = 0;
    } else if (letters.equals("gray")) {
      back = 204;
    }
    letters = "";  // Clear the variable
  } else if ((key > 31) && (key != CODED)) {
    // If the key is alphanumeric, add it to the String
    letters = letters + key;
  }
}
```

Controlling the flow

Programs written with `draw()` display frames to the screen as quickly as possible. The `frameRate()` function is used to set a limit on the number of frames that will display each second, and the `noLoop()` function can be used to stop `draw()` from looping. The additional functions `loop()` and `redraw()` provide more options when used in combination with the mouse and keyboard event functions.

If a program has been paused with `noLoop()`, running `loop()` resumes its action. Because the event functions are the only elements that continue to run when a program is paused with `noLoop()`, the `loop()` function can be used within these events to continue running the code in `draw()`. The following example runs the `draw()` function for two seconds each time a mouse button is pressed and then pauses the program after that time has elapsed.

26-09

```
int frame = 0;

void setup() {
  size(100, 100);
  frameRate(30);
}

void draw() {
  if (frame > 60) {                // If 60 frames since the mouse
    noLoop();                      // was pressed, stop the program
    background(0);                 // and turn the background black.
  } else {                         // Otherwise, set the background
    background(204);               // to light gray and draw lines
    line(mouseX, 0, mouseX, 100);  // at the mouse position
    line(0, mouseY, 100, mouseY);
    frame++;
  }
}

void mousePressed() {
  loop();
  frame = 0;
}
```

The `redraw()` function runs the code in `draw()` one time and then halts the execution. It's helpful when the display needn't be updated continuously. The following example runs the code in `draw()` once each time a mouse button is pressed.

```
void setup() {
  size(100, 100);
  noLoop();
}

void draw() {
  background(204);
  line(mouseX, 0, mouseX, 100);
}

void mousePressed() {
  redraw();  // Run the code in draw one time
}
```

Exercises

1. *Animate a shape to react when the mouse is pressed and when it is released.*
2. *Create two shapes and give each a different relation to the mouse. Design the behaviors of each shape so that it has one behavior when the mouse is moved and has another behavior when the mouse is dragged.*
3. *Write a program to update the display window* only *when a key is pressed.*

Notes

1. If a key is held down for an extended time, the code inside the `keyPressed()` block will run many times in a rapid succession. Most operating systems will take over and repeatedly call the `keyPressed()` function. The amount of time it takes to start repeating and the rate of repetitions will be different from computer to computer, depending on the keyboard perference settings.

Input 4: Mouse II

This unit introduces techniques for constraining and augmenting mouse data.

Syntax introduced:
`constrain(), dist(), abs(), atan2()`

The position of the cursor is a point within the display window that is updated every frame. This point can be modified and analyzed in relation to other elements to calculate new values. It's possible to constrain the mouse values to a specific range, calculate the distance between the mouse and another position, interpolate between two values, determine the speed of the mouse movement, and calculate the angle of the mouse in relation to another position. The code presented below enables each of these operations.

Constrain

The `constrain()` function limits a number to a range. It has three parameters:

```
constrain(value, min, max)
```

The `value` parameter is the number to limit, the `min` parameter is the minimum possible value, and the `max` parameter is the maximum possible value. This function returns the `min` number if the value parameter is less than or equivalent to `min`, returns the `max` number if the value parameter is more than or equivalent to `max`, and returns `value` without change if it's between the `min` and the `max`.

```
int x = constrain(35, 10, 90);  // Assign 35 to x
int y = constrain(5, 10, 90);   // Assign 10 to y
int z = constrain(91, 10, 90);  // Assign 90 to z
```

27-01

When used with the `mouseX` or `mouseY` variables, this function can set maximum and minimum values for the mouse coordinate data.

```
// Constrains the position of the ellipse to a region

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}
```

27-02

```
void draw() {
  background(0);
  // Limits mx between 35 and 65
  float mx = constrain(mouseX, 35, 65);
  // Limits my between 40 and 60
  float my = constrain(mouseY, 40, 60);
  fill(102);
  rect(20, 25, 60, 50);
  fill(255);
  ellipse(mx, my, 30, 30);
}
```

27-02 cont.

Distance

The `dist()` function calculates the distance between two coordinates. This value can be used to determine the cursor's distance from a point on screen in addition to its current position. The `dist()` function has four parameters:

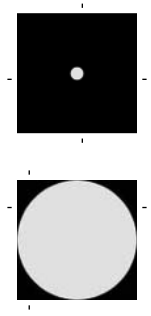
```
dist(x1, y1, x2, y2)
```

The *x1* and *y1* parameters set the coordinate of the first point, and the *x2* and *y2* parameters set the coordinate of the second point. The distance between the two points is calculated as a floating-point number and is returned:

```
float x = dist(0, 0, 50, 0);     // Assign 50.0 to x
float y = dist(50, 0, 50, 90);   // Assign 90.0 to y
float z = dist(30, 20, 80, 90);  // Assign 86.023254 to z
```

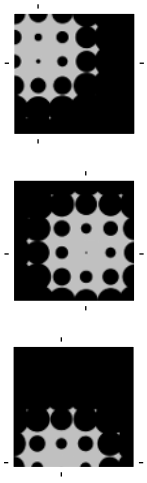
27-03

The value returned from `dist()` can be used to set the properties of shapes:



```
// The distance between the center of the display
// window and the cursor sets the diameter of the circle

void setup() {
  size(100, 100);
  smooth();
}

void draw() {
  background(0);
  float d = dist(width/2, height/2, mouseX, mouseY);
  ellipse(width/2, height/2, d*2, d*2);
}
```

27-04

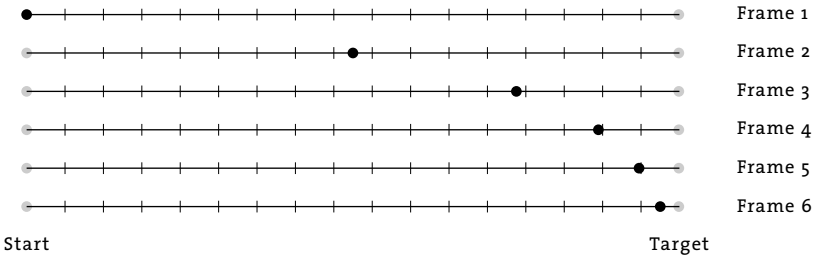


27-05

```
// Draw a grid of circles and calculate the
// distance to each to set the size

float maxDistance;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  fill(0);
  maxDistance = dist(0, 0, width, height);
}

void draw() {
  background(204);
  for (int i = 0; i <= width; i += 20) {
    for (int j = 0; j <= height; j += 20) {
      float mouseDist = dist(mouseX, mouseY, i, j);
      float diameter = (mouseDist / maxDistance) * 66.0;
      ellipse(i, j, diameter, diameter);
    }
  }
}
```

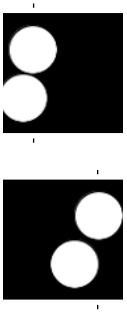
Easing

Easing, also called interpolation, is a technique for moving between two points. By moving a fraction of the total distance each frame, a shape can decelerate (or accelerate) as it approaches a target location. This diagram shows what happens when a point always moves half of the way between its current position and the destination:



As the shape approaches the target position, the distance moved each frame decreases; therefore, the shape slows down. In the following example the `x` variable is the current horizontal position of the circle and the `targetX` variable is the destination position.

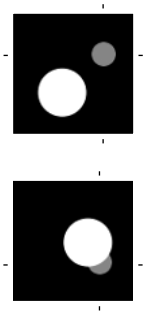
The `easing` variable sets the fraction of the distance between the circle's current position and the position of the mouse that the circle moves each frame. The value of this variable changes how quickly the circle will reach the target. The value must always be between 0.0 and 1.0, and numbers closer to 0.0 cause the easing to take more time. An `easing` value of 0.5 means the circle will move half the distance each frame and an easing value of 0.01 means the circle will move one hundredth of the distance each frame. The top ellipse is drawn at the `targetX` position and the bottom ellipse is drawn at the interpolated position.



27-06

```
float x = 0.0;
float easing = 0.05;  // Numbers 0.0 to 1.0

void setup() {
  size(100, 100);
  smooth();
}

void draw() {
  background(0);
  float targetX = mouseX;
  x += (targetX - x) * easing;
  ellipse(mouseX, 30, 40, 40);
  ellipse(x, 70, 40, 40);
}
```

To apply the same principle simultaneously to the x- and y-coordinate values, add an additional set of variables and test for the distance for both. In this example the small circle is always at the target position controlled by the cursor, and the large circle is positioned with the easing equation.



27-07

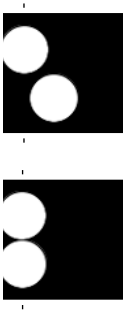
```
float x = 0;
float y = 0;
float easing = 0.05;  // Numbers 0.0 to 1.0

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(0);
  float targetX = mouseX;
  float targetY = mouseY;
```

```
  x += (targetX - x) * easing;
  y += (targetY - y) * easing;
  fill(153);
  ellipse(mouseX, mouseY, 20, 20);
  fill(255);
  ellipse(x, y, 40, 40);
}
```

27-07 cont.

The previous two examples continue to make the calculation for the circle position even after it has reached the destination. This is inefficient, and if there were thousands of circles all easing between positions, it would slow down the program. To stop the calculations when they are no longer necessary, test to see if the target position and destination position are the same and stop the calculation if they are.

The following example introduces the `abs()` function for taking the absolute value of a number. This is necessary because the values used in easing are either negative or positive depending on whether the position is to the left or to the right of the target. An `if` structure is used to update position only if it is not at the same pixel as the target.



```
float x = 0.0;
float easing = 0.05;  // Numbers 0.0 to 1.0

void setup() {
  size(100, 100);
  smooth();
}

void draw() {
  background(0);
  float targetX = mouseX;
  // Distance from position and target
  float dx = targetX - x;
  // If the distance between the current position and the
  // destination is greater than 1.0, update the position
  if (abs(dx) > 1.0) {
    x += dx * easing;
  }
  ellipse(mouseX, 30, 40, 40);
  ellipse(x, 70, 40, 40);
}
```

27-08

Speed

Calculate the mouse speed by comparing the current position with the previous position. This is done by using the `dist()` function with the `mouseX`, `mouseY`, `pmouseX`, and `pmouseY` values as the parameters. The following example calculates the speed of the mouse and converts this value into the size of an ellipse.

27-09

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  background(0);
  float speed = dist(mouseX, mouseY, pmouseX, pmouseY);
  float diameter = speed * 3.0;
  ellipse(50, 50, diameter, diameter);
}
```

The previous examples show the instantaneous speed of the mouse. The numbers produced are extreme—they jump between zero and large values from one frame to the next. The easing equation from code 27-06 (p. 240) can be used to increase and decrease speed smoothly. The following example demonstrates how to apply the easing equation to this context. The top bar is the instantaneous speed and the bottom bar is the eased speed.

27-10

```
float speed = 0.0;
float easing = 0.05;  // Numbers 0.0 to 1.0

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  background(0);
  float target = dist(mouseX, mouseY, pmouseX, pmouseY);
  speed += (target - speed) * easing;
  rect(0, 33, target, 17);
  rect(0, 50, speed, 17);
}
```

Orientation

The `atan2()` function is used to calculate the angle from any point to the coordinate (0,0). It has two parameters:

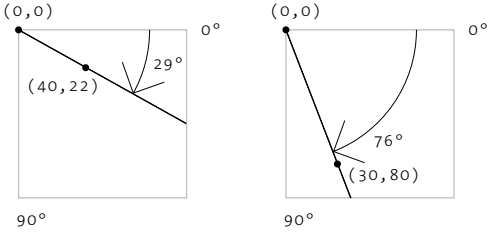
```
atan2(y, x)
```

The *y* parameter is the y-coordinate of the point from which to find the angle, and the *x* parameter is the x-coordinate of the point. Angle values are returned in radians in the range of π to −π . Note that the order of the *y* and *x* parameters are reversed from other functions we've seen.

27-11

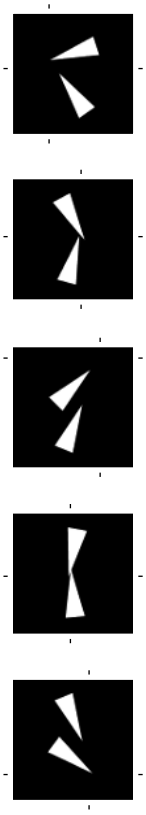
```
// The angles increase as the mouse moves from the upper-right
// corner of the screen to the lower-left corner

void setup() {
  size(100, 100);
  frameRate(15);
  fill(0);
}

void draw() {
  float angle = atan2(mouseY, mouseX);
  float deg = degrees(angle);
  println(deg);
  background(204);
  ellipse(mouseX, mouseY, 8, 8);
  rotate(angle);
  line(0, 0, 150, 0);
}
```

The code is explained with these images:



To calculate `atan2()` relative to another point instead of (0,0), subtract the coordinates of that point from the *y* and *x* parameters. Use the *`translate()`* function to position the elements on screen, rotate or orient them using the result of `atan2()`, and use

pushMatrix() and popMatrix() to isolate the transformations. This procedure is discussed in depth in Transform 1 (p. 133) and Transform 2 (p. 137).



27-12

```
// Rotate the triangles so they always point
// to the cursor

float x = 50;
float y1 = 33;
float y2 = 66;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  background(0);

  // Top triangle
  float angle = atan2(mouseY-y1, mouseX-x);
  pushMatrix();
  translate(x, y1);
  rotate(angle);
  triangle(-20, -8, 20, 0, -20, 8);
  popMatrix();
  pushMatrix();

  // Bottom triangle
  float angle2 = atan2(mouseY-(y2), mouseX-x);
  translate(x, y2);
  rotate(angle2);
  triangle(-20, -8, 20, 0, -20, 8);
  popMatrix();
}
```

Exercises

1. *Change the properties of a shape based on the cursor's distance to its center.*
2. *Use the easing equation to create a shape that acts lazy.*
3. *Using the techniques introduced in this unit, create three shapes that each follow the mouse in a different way.*

Input 5: Time, Date

This unit introduces using the current time and date as inputs.

Syntax introduced:
`second(), minute(), hour(), millis(), day(), month(), year()`

Humans have a relative perception of time, but machines attempt to keep precise, regular time. Previous civilizations used sundials and water clocks to visualize the passage of time; today, most people use the digital numeric clock and the twelve-hour circular clock with a minute, second, and hour hand. Each of these representations reflects their technology. A numeric digital readout is appropriate for a digital timekeeping mechanism in need of an inexpensive display. A timekeeping mechanism built from circular gears lends itself to a circular presentation of time. The tower-sized clocks of the past with enormous gears and weights have evolved and shrunk into devices so inexpensive and abundant that digital devices such as microwave ovens, mobile phones, and computers all display the time and date.

Reading the current time and date into a program opens an interesting area of exploration. Knowledge of the current time makes it possible to write a program that changes its colors every day depending on the date, or a digital clock that plays an animation every hour. The ability to input time and date information enables software tools for remembering, reminding, and informing and creates the potential for whimsical time-based events.

Seconds, Minutes, Hours

Processing programs can read the value of the computer's clock. The current second is read with the `second()` function, which returns values from 0 to 59. The current minute is read with the `minute()` function, which also returns values from 0 to 59. The current hour is read with the `hour()` function, which returns values in the 24-hour time notation from 0 to 23. In this system midnight is 0, noon is 12, 9:00 a.m. is 9, and 5:00 p.m. is 17. Run this program to see the current time:

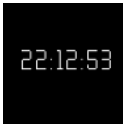
28-01

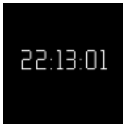
```
int s = second();  // Returns values from 0 to 59
int m = minute();  // Returns values from 0 to 59
int h = hour();    // Returns values from 0 to 23
println(h + ":" + m + ":" + s);  // Prints the time to the console
```

Placing these functions inside `draw()` allows the time to be read continuously. This example reads the current time and updates the text area with the passage of each second:

```
int lastSecond = 0;                                                    28-02

void setup() {
  size(100, 100);
}

void draw() {
  int s = second();
  int m = minute();
  int h = hour();
  // Only prints once when the second changes
  if (s != lastSecond) {
    println(h + ":" + m + ":" + s);
    lastSecond = s;
  }
}
```

You can create a clock with a numerical display by using the `text()` function to draw the numbers to the display window. The `nf()` function (p. 422) is used to space the numbers equally from left to right. Single digit numbers are padded on their left with a zero so all numbers occupy a two-digit space at all times.





```
PFont font;                                                            28-03

void setup() {
  size(100, 100);
  font = loadFont("Pro-20.vlw");
  textFont(font);
}

void draw() {
  background(0);
  int s = second();
  int m = minute();
  int h = hour();
  // The nf() function spaces the numbers nicely
  String t = nf(h,2) + ":" + nf(m,2) + ":" + nf(s,2);
  text(t, 10, 55);
}
```

There are many different ways to express the passage of time. In the next example, horizontal lines mark the current second, hour, and minute. The left edge of the display window is 0 and the right edge is the maximum for each time component. Because the values from the time functions range from 0 to 59 and from 0 to 23, they are modified to

all have the range of 0 to 99. Each time value is divided by its maximum value and then multiplied by 100.0.

28-04

```
void setup() {
  size(100, 100);
  stroke(255);
}

void draw() {
  background(0);
  float s = map(second(), 0, 60, 0, 100);
  float m = map(minute(), 0, 60, 0, 100);
  float h = map(hour(), 0, 24, 0, 100);
  line(s, 0, s, 33);
  line(m, 34, m, 66);
  line(h, 67, h, 100);
}
```

These normalized time values can also be used to simulate the second, minute, and hour hands on a traditional clock face. In this case, the values are multiplied by 2π to display the time as points around a circle. Because the `hour()` function returns values in 24-hour time, the program converts the hour value to 12-hour time scale by calculating the hour % 12 (the % symbol is introduced on p. 45).

28-05

```
void setup() {
  size(100, 100);
  stroke(255);
}

void draw() {
  background(0);
  fill(80);
  noStroke();
  // Angles for sin() and cos() start at 3 o'clock;
  // subtract HALF_PI to make them start at the top
  ellipse(50, 50, 80, 80);
  float s = map(second(), 0, 60, 0, TWO_PI) - HALF_PI;
  float m = map(minute(), 0, 60, 0, TWO_PI) - HALF_PI;
  float h = map(hour() % 12, 0, 12, 0, TWO_PI) - HALF_PI;
  stroke(255);
  line(50, 50, cos(s) * 38 + 50, sin(s) * 38 + 50);
  line(50, 50, cos(m) * 30 + 50, sin(m) * 30 + 50);
  line(50, 50, cos(h) * 25 + 50, sin(h) * 25 + 50);
}
```

In addition to reading the current time, each Processing program counts the time passed since the program started. This time is stored in milliseconds (thousandths of a second). Two thousand milliseconds is 2 seconds, and 200 milliseconds is 0.2 seconds. This number is obtained with the `millis()` function and can be used to trigger events and calculate the passage of time:

28-06

```
// Uses millis() to start a line in motion three seconds
// after the program starts

int x = 0;

void setup() {
  size(100, 100);
}

void draw() {
  if (millis() > 3000) {
    x++;
  }
  line(x, 0, x, 100);
}
```

The `millis()` function returns an `int`, but it is sometimes useful to convert it to a `float` that represents the number of seconds elapsed since the program started. The resulting number can be used to control the sequence of events in an animation.

28-07

```
int x = 0;

void setup() {
  size(100, 100);
}

void draw() {
  float sec = millis() / 1000.0;
  if (sec > 3.0) {
    x++;
  }
  line(x, 0, x, 100);
}
```

Date

Date information is read in a similar way as the time. The current day is read with the `day()` function, which returns values from 1 to 31. The current month is read with the `month()` function, which returns values from 1 to 12 where 1 is January, 6 is June, and 12 is December. The current year is read with the `year()` function, which returns the four-digit integer value of the present year. Run this program to see the current date in the console:

28-08

```
int d = day();    // Returns values from 1 to 31
int m = month();  // Returns values from 1 to 12
int y = year();   // Returns four-digit year (2007, 2008, etc.)
println(d + " " + m + " " + y);
```

The following example checks to see if it is the first day of the month and prints the message "Welcome to a new month." to the console if it is the first day of the month.

28-09

```
void draw() {
  int d = day();  // Values from 1 to 31
  if (d == 1) {
    println("Welcome to a new month.");
  }
}
```

The following example runs continuously and checks to see if it is New Year's Day. If so, the message "Today is the first day of the year!" is printed to the console.

28-10

```
void draw() {
  int d = day();    // Values from 1 to 31
  int m = month();  // Values from 1 to 12
  if ((d == 1) && (m == 1)) {
    println("Today is the first day of the year!");
  }
}
```

Exercises

1. *Make a simple clock to run an animation for two seconds at the beginning of each minute.*
2. *Create an abstract clock that communicates the passage of time through graphical quantity rather than numerical symbols.*
3. *Write a program to draw images to the display window corresponding to specific dates (e.g., display a pumpkin on Halloween).*

```
void Draw() (
  background(126)
  ellipse(mouseX. mousey, 33
}
```

```
void draw() {
  background(126);
  ellipse(mouseX, mouseY, 33
}
```

Development 2: Iteration, Debugging

This unit discusses the iterative software development process and the activity of debugging code.

The programs included up to this point in the book have been short. Programs of this length can be written without much forethought, but planning becomes important when writing longer programs. The extent of the planning will be up to the programmer, but one aspect of programming is always the same: large, complex programs must be divided into series of short, simpler programs. Learning how to divide programs into manageable parts takes time and experience. As the scope of a program grows, the number of decisions involved in writing it multiplies. Making changes to a program, evaluating the result, and then making additional changes is an iterative process. Like a project in any medium, software improves through many cycles of changes and evaluations.

Longer programs also present a higher likelihood of mistakes. The flow of logic and data becomes less obvious in a larger program, and the errors—known as bugs—introduced are more subtle. Learning to track down and fix errors is an important skill in writing software.

Iteration

There are many different models for software development, but they all contain elements of analysis, synthesis, and evaluation. A continuous cycle of synthesis and evaluation is the core of the iterative process. Every project demands variations on each of these stages, but the purpose of each remains consistent. A more detailed description of each stage illuminates how they interact:

Analysis
Analysis leads to an understanding of the software—its function, audience, and purpose. This stage can involve months of research or mere seconds of consideration, resulting in a proposal, project description, or other means of communicating the project to others.

Synthesis
The goals and concepts that emerge from the analysis are realized through synthesis. Early steps in synthesis often include paper sketches, followed by software sketches, and then refinement of the finished software. The results of this stage are evaluated, edited, and augmented with additional synthesis until the software is finished.

Evaluation
The results of the synthesis phase are evaluated in relation to the analysis to determine what remains to be done. Is the project complete or is another round of synthesis needed? What improvements can be made? What is working and what needs to be fixed? Depending on the nature of the project, the evaluation sometimes returns to analysis and the goals of the project are modified.

Programs change quickly, and sometimes the programmer prefers an earlier version. Save multiple versions of the sketch while working so it's always possible to return to a previous iteration of the code. Simply select "Save As" from the File menu to save a new version of the program with a different name. The "Archive Sketch" option from the Tools menu saves the code and all additional media for the current sketch inside a ZIP archive with the name of the current sketch and the date. Saving multiple versions of a sketch ensures that older, working examples of the code remain intact.

Debugging

When a person first starts programming, errors (bugs) occur frequently; learning how to find and fix (debug) them is an important part of learning to program. In *The Practice of Programming*, the authors explain: "Good programmers know that they spend as much time debugging as writing so they try to learn from their mistakes. Every bug you find can teach you how to prevent a similar bug from happening again or to recognize it if it does." [1]

Some bugs reveal themselves when the Run button is pressed, as they prevent the program from starting. Other bugs appear while the program is running, causing the program to stop. The message area (p. 8) turns red and reveals a summary of the problem. Sometimes the bug message text is too long to fit in this area, but the full message always appears in the console. The Processing environment always tries to highlight the line where the bug occurs, but since the bug may be the result of something that happened earlier in the program, the error does not always appear on the highlighted line. The highlighted line is usually related to the error, but perhaps not in an obvious way.

Not all bugs stop a program from running. Errors in logic or problems with equations are sometimes more difficult to find because they don't stop the program.

Fixing bugs is one of the more difficult and less satisfying aspects of programming. Sometimes they are obvious and quick to fix, but sometimes it can take hours. Finding a bug is like solving a mystery. It's necessary to search through the code to find clues in pursuit of the culprit. Try the following:

Scrutinize the newest code
If the program is constructed step by step, the bug is often in the newest code or is linked to it. Check these areas for bugs first.

Check related code
Sometimes a bug may linger within a program for a long time because the line containing the bug is not run. When code is introduced that runs a line with a bug, or when the value of a variable changes so that code within a previously unused `if` or `for` structure is run, the bug will reveal itself.

Display output
Displaying the data produced by a program while it's running can expose problems and lead to a better understanding of the code in general. The `println()` function can be used to display data as text to the console. This technique can answer questions about the status of a variable and can be used to check whether a specific line or block of code is running. The data can also be represented as positions or colors in the program's display.

Isolate the problem
It's often difficult to find a bug within a large program. If possible, try to reduce the problem to its essence. Is it possible to reproduce the bug by running only a few lines of code or a much simpler program?

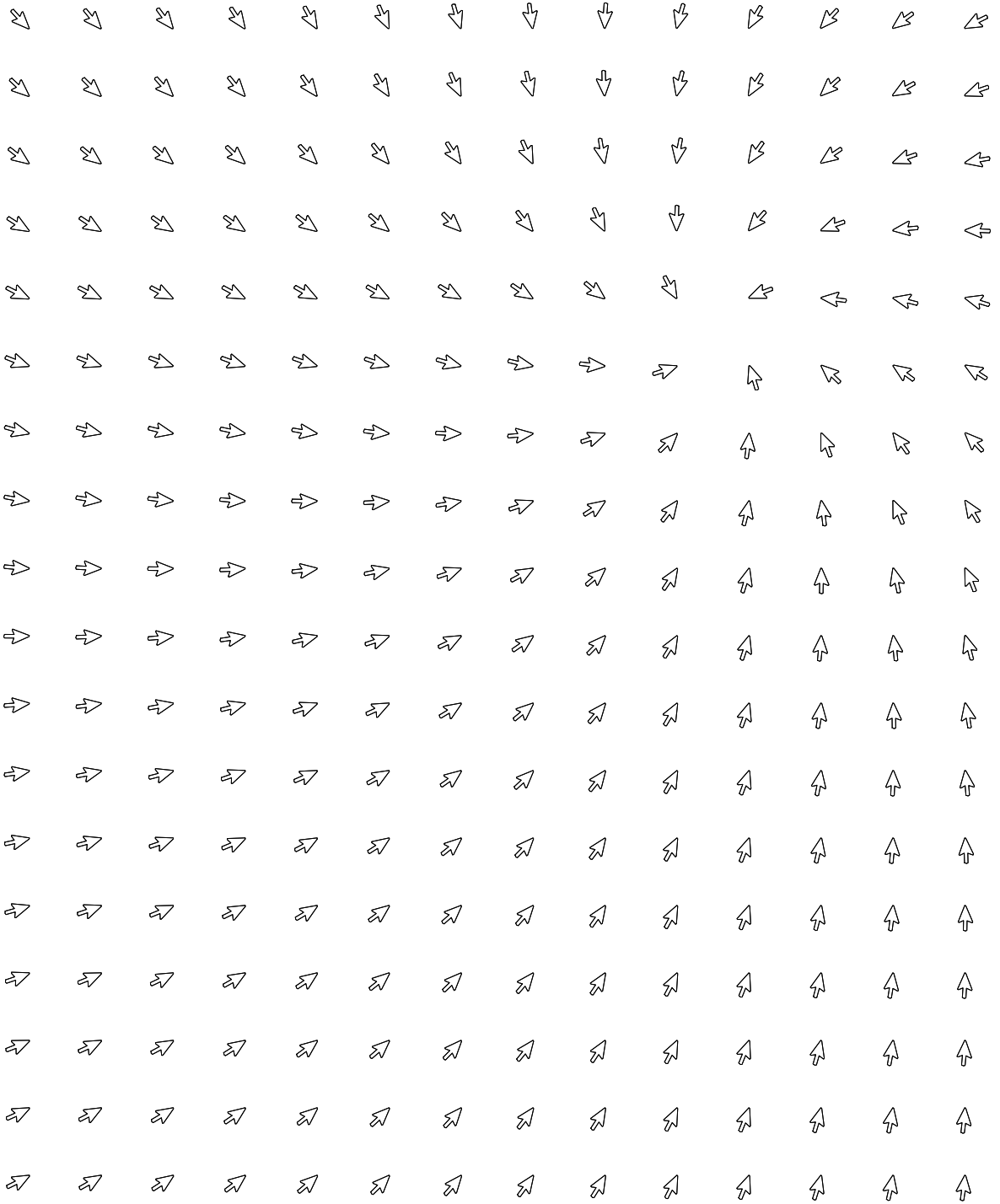
Learn from previous bugs
All programmers—new and experienced—inadvertently introduce bugs into their code. The hardest time to find a particular bug is usually during its first occurrence. Learn from previous mistakes to avoid the same bug in the future.

Take a break
Sometimes the best way to fix a bug is to take a break. After hours of programming, the perspective gained from a diversion or rest can bring clarity.

As with all software, there are bugs in Processing, and some are added and removed with each release of the software. For the most current information about bugs in the Processing software, read the Frequently Asked Questions (FAQ), accessible from the Help menu. A complete list can be found at http://dev.processing.org/bugs. This website can be searched for known bugs and used to report new ones.

Notes

1. Brian W. Kernighan and Bob Pike, *The Practice of Programming* (Addison-Wesley, 1999), p. 117.



Synthesis 2: Input and Response

This unit presents examples of synthesizing concepts from Structure 2 to Development 2.

The previous units introduced programs that run continuously, functions, parameterized form, mouse input, keyboard input, events, and reading the time. This synthesis unit emphasizes the concept of response. Within the domain of these programs, response is an action that corresponds to an input, a stimulus. For example, if a key is pressed, how does the program react? If the mouse is moved, how does the program respond?

The artist Myron Krueger coined the phrase "Response is the medium!"[1] He was primarily interested in how his work responded to people. He focused on the aesthetics of response, rather than the aesthetics of the image or motion. Kruger pointed out a number of ways people can relate to a responsive system such as a work of software. An individual can have a dialog with the system or can be a protagonist in an open-ended narrative or a participant in a game. The system can be an amplifier, a space to explore, or an instrument.

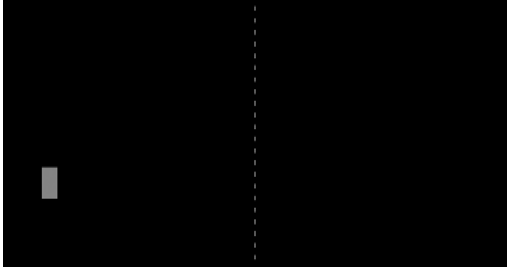
The act of creating responsive software consists of relating an input to possible outputs. Unexpected juxtapositions between the stimulus and response can engage the mind of the participant. Predictable and repetitive relationships between an action and the reaction can be tedious. In soccer or chess, not being sure of the opponent's first move keeps the activity engaging. This uncertainty creates a tension and the potential that no game will be played the same way as another.

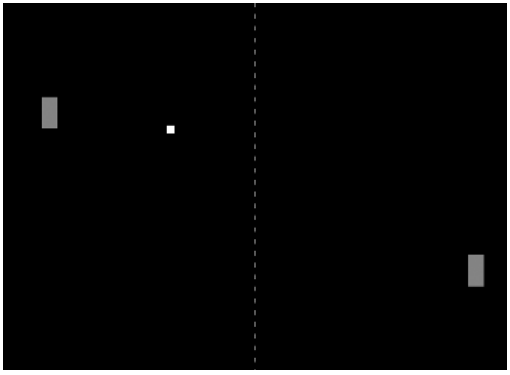
The four programs introduced in this unit each offer a different perspective on the concept of response. The first is a simulation of a classic video game, the second questions the assumptions we make about how a mouse works, the third uses the time as its input to create an abstract clock, and the fourth presents a new way to think about typing and organizing lines of text.

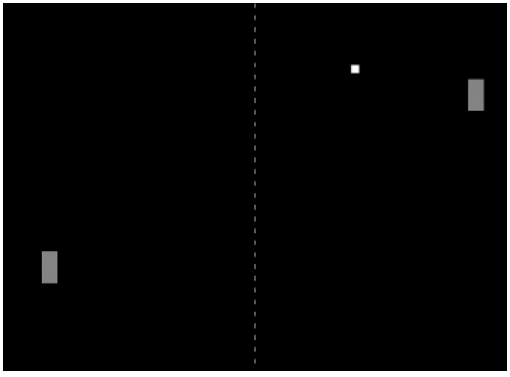
The four programs presented here were written by different programmers. Unlike most of the other examples in the book, which have been written in a similar style, each of these programs reflects the personal programming style of its author. Learning how to read programs written by other people is an important skill.
The software featured in this unit is longer than the brief examples that fill this book. It's not practical to print it on these pages, but the code is included in the Processing code download at www.processing.org/learning.

Notes

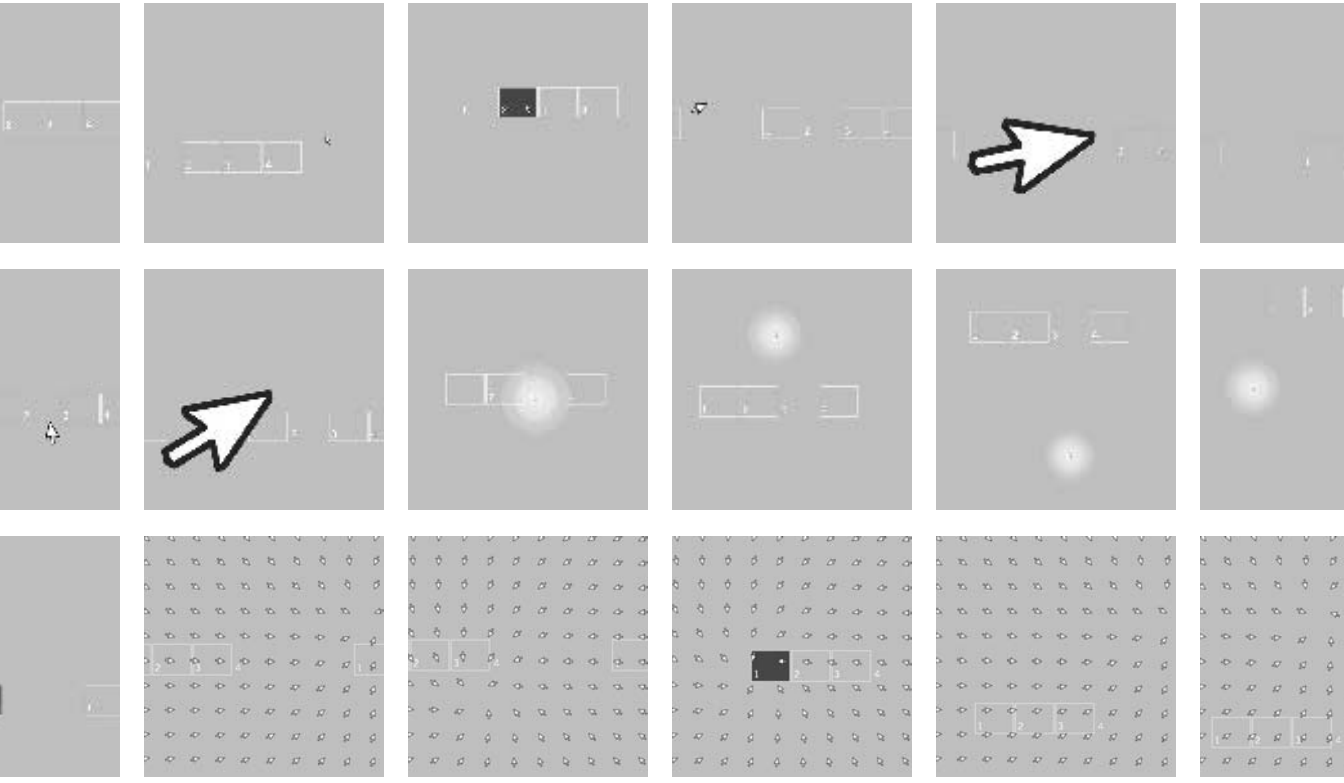
1. Myron Krueger, *Artificial Reality II* (Addison-Wesley, 1991), p.85.







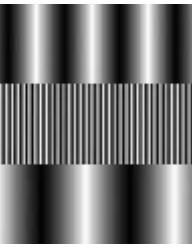
Tennis. Video games began to appear more than thirty years ago. Pong, a table tennis game developed in the early 1970s, was the first game to catch the attention of the public. The square "ball" moves across the screen, and the objective is to keep the ball on the screen by hitting it with the paddle. This simulated version of the game was written for one player. The paddle on the right is positioned at the y-coordinate of the mouse and the one on the left is set to the inverse position. The player changes the angle of the ball by moving the paddle up or down as it strikes the ball. Video games like Pong were successful because they were highly responsive, despite extremely minimal imagery.



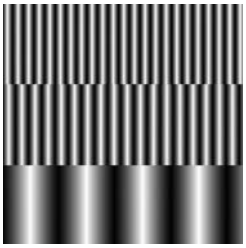
Cursor. After using computers for years, a programmer can take for granted the way the cursor and mouse are linked. Custom software, however, can change this relationship. This program presents the relation between the mouse and cursor in four different ways. In the first mode, the cursor behaves as we expect. Select an alternative mode by clicking on one of the boxes. In the second mode, the size and orientation of the cursor is determined by its speed and direction. In the third mode, the cursor is lazy and responds to the mouse slowly. The fourth mode multiplies the cursor into a matrix, and each individual element points to the actual position determined by the mouse.

Program written by Peter Cho (www.typotopo.com)

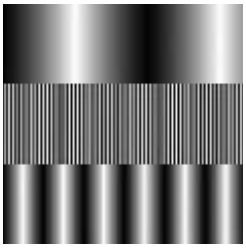
Typing. Unlike those produced by a word processor, the lines of text in this program can be positioned anywhere on the screen. Clicking and dragging the mouse changes the size and the angle. The program allows a maximum of five lines to be edited at once. Press the Enter or Return key to switch between lines and press the Backspace key to remove a letter from the end of the line. Move the mouse across the screen to change the position of the current lines.



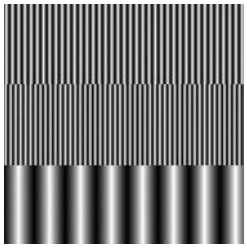
04:18:22



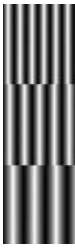
05:56:01



07:48:39



09:10:44



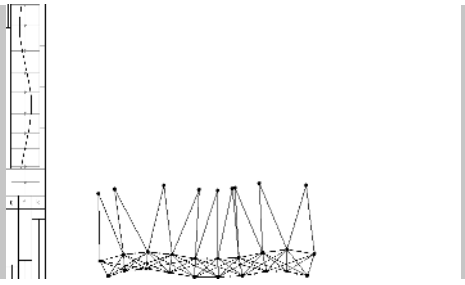
Banded Clock. The images on this page tell the time with visual patterns, rather than with numbers. The `sin()` function generates bands based on the values read from the computer's clock. Count the number of bands on each line to determine each time component. Like code 28-04 (p. 247), the top row is seconds, the middle is minutes, and the bottom is hours.

Program written by Golan Levin (www.flong.com)

Hektor (far right) paints the William Morris "Compton" design from 1896 onto a wall.
Image courtesy of **Jürg Lehni and Uli Franke**. Realized in collaboration with Goodwill.

Interviews 2: Software, Web

Ed Burton. *Sodaconstructor*
Josh On. *They Rule*
Jürg Lehni. *Hektor* and *Scriptographer*
Auriea Harvey and Michaël Samyn. *The Endless Forest*



Sodaconstructor *(Interview with Ed Burton)*

Creators	Ed Burton and Soda Creative Ltd.
Year	1998
Medium	Software
Software	Java
URL	www.sodaplay.com/constructor, www.sodaplay.com/zoo

What is *Sodaconstructor*?

Sodaconstructor *is a virtual two-dimensional construction kit equipped with point masses and springs. It contains a very simple simulation of Hooke's law, which stipulates that the force applied by a spring is proportional to its extension. Structures can be drawn, simulated, and manipulated in a surprisingly tactile manner. Usually springs have a fixed rest length; however, in* Sodaconstructor *selected springs can be transformed into "muscles" whose rest length oscillates over time in response to an on-screen sine wave. By modifying variables such as gravity and friction and carefully arranging masses, springs, and muscles (typically through a playful iterative process of trial and error) all manner of perambulating automata, animated drawings, or pulsating abstract compositions can be constructed.*

An online community has been exploring Sodaconstructor *for over seven years, contributing their creations to the* Sodazoo *where we witness an amazing menagerie of hundreds of thousands of creations that far surpass my own ability to use my software and that continue to delight and surprise.*

Why did you create *Sodaconstructor*?

Sodaconstructor *was an invented programming exercise to teach myself the Java programming language. I thought of a toy that I wanted to play with and set myself the hurdle of having to learn enough Java to build it before I could play with it. I was also interested in dynamical systems, viewing behavior as something that emerges over time through a process of feedback.* Sodaconstructor *was an attempt to make a simple toy based on this principle.*

I neither intended nor anticipated that anyone else would want to play with Sodaconstructor*; it was a piece of pure personal play for the purpose of my own learning. My immediate reward was to enjoy making a dozen or so crude creatures such as "daintywalker" and "amoeba" while developing* Sodaconstructor *and playing with it for a relatively short time after its initial completion.*

Two years later, in the summer of 2000, there was a huge, seemingly spontaneous explosion in Sodaconstructor *traffic. Frustrated that we couldn't preserve the results of this surprising and sustained surge, Soda developed the* Sodazoo*, a database-driven gallery where visitors could save and share their creations. As our database steadily filled with over half a million creations, many of which are exquisite in their union of engineering sophistication and graceful beauty, my interest shifted from creating creatures with my own software to facilitating creativity in others. The* Sodazoo *now sustains a creative ecology that evolves through an iterative process of peer inspiration to achieve ever-greater feats of quality and diversity.*

Sodaplay *also has a sister project,* Sodarace.net*, in which* Sodaconstructor *models can be raced over two-dimensional terrains. However it's not only humans who get to design the*

Sodaconstructor, 1998. Images courtesy of Soda Creative Ltd.

contenders. It is also possible for automated optimization software such as genetic algorithms to be plugged into the race, iteratively submitting machine-generated models and receiving race results in order to breed race winners. The intention of Sodarace is not only to have fun but also to expose a wider audience to some of the principles and practice of engineering and artificial intelligence research.

Soda is now hard at work developing the next major evolution of Sodaplay. *While the* Sodaplay *community continues to reward me with surprising new* Sodaconstructor *creations, I have become increasingly concerned that they are constrained by the software frame that I have given them. They can create original content within my application but they cannot modify or extend its user interface or behavior in any way. To respond radically to this we are rebuilding* Sodaconstructor *within an entirely new* Sodaplay *that not only exposes application source code, but also makes application user interfaces and behaviors both malleable and extensible. Sodaplay used to be the home of a single creative tool,* Sodaconstructor. *In the future it will be a tool for creating many creative tools. Our aspiration is to enable the* Sodaplay *community to expand its creative ecology to encompass not only the content, but also the software and context that creates the content.*

What software tools were used?

Sodaconstructor *is a Java applet, an interactive application that runs in a Web browser. For two years that's all* Sodaconstructor *was—people could play and construct, but their creations were lost when they closed their browsers. Two years later we built the Sodazoo to give a home to our users' creations using a Java servlet to connect to an Oracle database. Now Sodaplay is growing into a full-fledged web application using the Spring Framework (that's Spring as in* www.springframework.org*, a layered Java/J2EE application framework and nothing to do with the bouncy simulated springs inside* Sodaconstructor*!).*

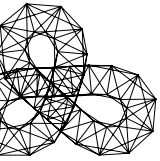
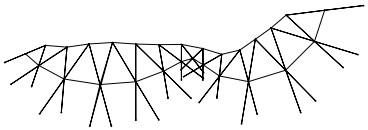
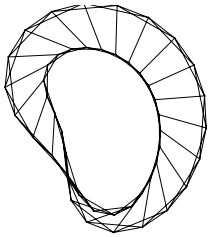
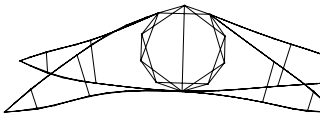
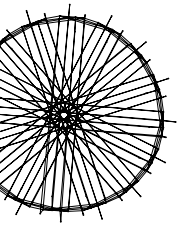
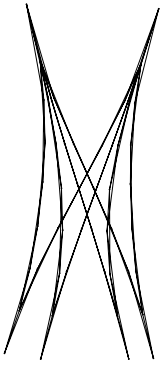
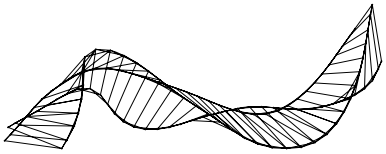
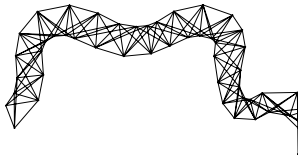
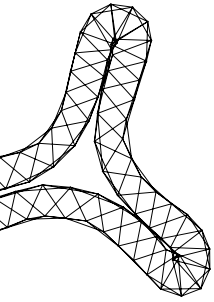
Why did you use these tools?

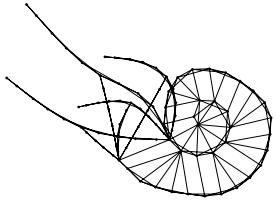
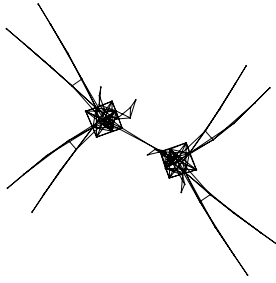
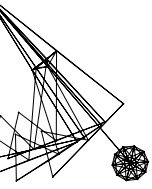
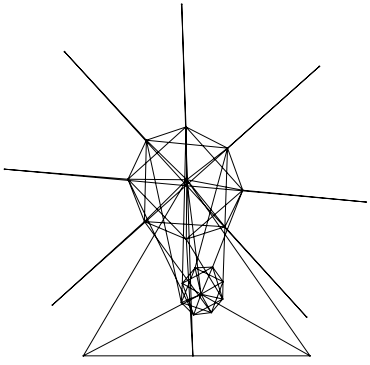
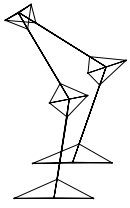
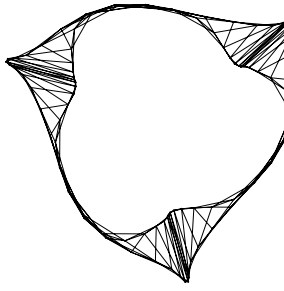
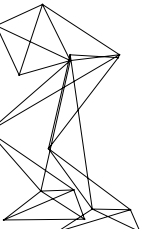
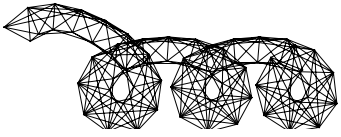
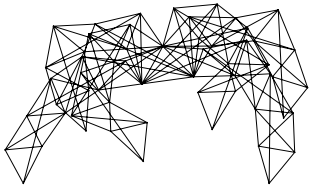
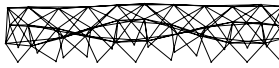
I'd been playing with programming since my early teens. After first learning BASIC on a 48K Sinclair Spectrum and BBC Acorn Archimedes, then C and later C++. I chose to learn Java because it was flexible enough to realize anything I could imagine in the previous languages I'd worked with but promised one distinctive advantage: it was easy to distribute to users on multiple platforms (Mac OS, Windows, etc.) over the Internet.

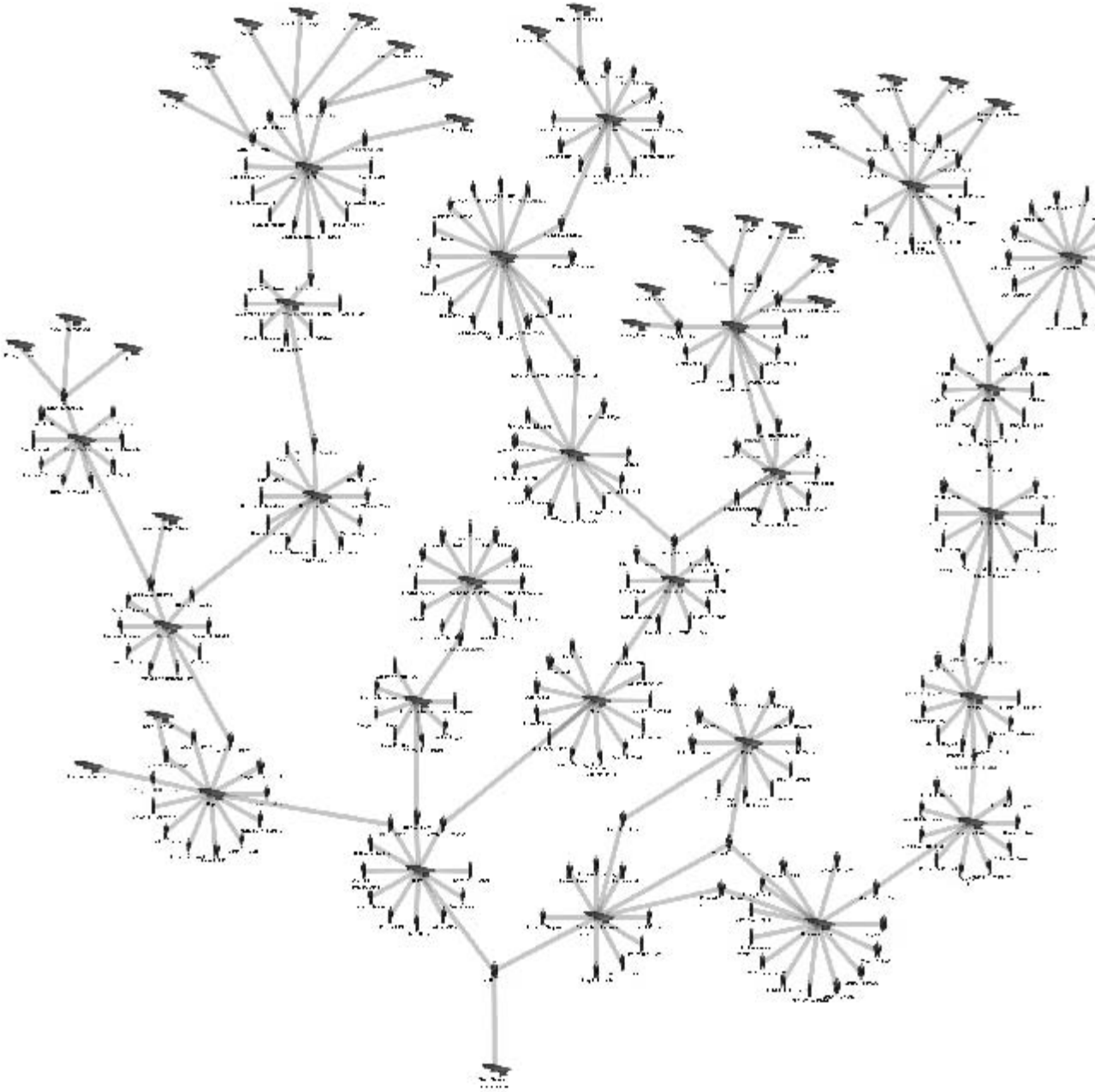
Why do you choose to work with software?

I tend to be more interested in process than product. Rather than create singular static forms I want to create processes that may result in countless potential forms emerging over time. My motivation is to be happily surprised by the process of emergence. It may sound counterintuitive that programming can be surprising. Computer programs are constructed from deterministic logical operations, after all. However, even the simplest operations can have unexpected consequences, especially when executed with feedback and interactivity. By using programming to harness feedback and interactivity it is possible to be repeatedly rewarded with emergent phenomena and happy surprises.

Models from the *SodaZoo*. Images courtesy of Soda Creative Ltd.



They Rule *(Interview with Josh On)*

Creators	Josh On, Amy Balkin, and Amy Franceschini
Year	2001, 2004
Medium	Web
Software	Flash, PHP, MySQL
URL	www.theyrule.net

What is *They Rule*?

They Rule *aims to provide a glimpse of some of the relationships of the U.S. ruling class. It takes as its focus the boards of some of the most powerful U.S. companies, which share many of the same directors. Some individuals sit on 5, 6, or 7 of the top 500 companies. It allows users to browse through these interlocking directories and run searches on the boards and companies. A user can save a map of connections complete with their annotations and Email links to these maps to others.* They Rule *is a starting point for research into these powerful individuals and corporations.*

Why did you create They Rule?

America is a class-divided society. There is no greater contradiction in our society than the fact that the majority of the people who do the work are not the ones who reap the benefit. In 1998 the top 1 percent of the population owned 38 percent of the wealth; the top 5 percent owned over 60 percent.[1] *That was the situation in the "boom years." This inequality might be overlooked as long as the people at the bottom end of the scale have what they need. They don't. According to the* CIA World Fact Book, *12.5 percent of Americans live below the poverty line. There is enough to go around; we just have a system that doesn't let it flow.*

A few companies control much of the economy, and oligopolies exert control in nearly every sector of the economy. The people who head up these companies swap on and off the boards from one company to another, and in and out of government committees and positions. These people run the most powerful institutions on the planet, and we have almost no say in who they are. This is not a conspiracy. They are proud to rule, yet these connections of power are not always visible to the public eye.

Karl Marx once called this ruling class a "band of hostile brothers." They stand against each other in the competitive struggle for the continued accumulation of their capital, but they stand together as a family supporting their interests in perpetuating the profit system as a whole. Protecting this system can require the cover of a "legitimate" force—and this is the role that is played by the state. An understanding of this system cannot be gleaned from looking at the interpersonal relations of this class alone, but rather how they stand in relation to other classes in society. Hopefully They Rule *will raise larger questions about the structure of our society and in whose benefit it is run.*

I wanted They Rule *to provide a starting point for getting some facts.* They Rule *graphically reveals this one surface reality of an interconnected ruling class, but it also encourages visitors to dig deeper. It is easy to run a search on companies and individuals straight from the site. It is not uncommon for the first result in an Internet search engine query on a board member to come up with their name in connection with a government committee or advisory board, or*

Map of the Clear Channel board of directors and the connections. Created with *They Rule*. Image courtesy of Josh On.

even to reveal that they were in government for a time. The people in They Rule *include an ex-president, an ex-secretary of the Treasury, and many ex-members of Congress. The ongoing Enron scandal, which sparked much activity on the site, revealed just how closely tied the state is to the corporate world. As Marx put it, the state is "the executive committee of the ruling class." Hopefully* They Rule *can help us confirm (or deny) this.*

Far from being an exhaustive exposé of the ruling class, They Rule *shows only the smallest section of the relationships of control. It does not show the patterns of ownership or wealth, the cultural ties, the institutional and social connections that these people have with each other and others in their class that do not enter the map. Neither does it show the source of their power, the exploitation of labor and nature. It is a challenge that stands before us to illustrate some of these relations in a way that is compelling and revealing.*

What software tools were used?

I used a variety of tools, the main one being Flash, which I used for the client side. I used PHP, MySQL, and PHPMyAdmin to build the back end and the databases. I used 3d Studio Max to make the little people. I used Photoshop to export them for use in Flash. I used Textpad to do lots of data formatting.

Why did you use these tools?

The main question was what to use for the front end. It may have been possible to build They Rule *with HTML and JavaScript—but it would have been a nightmare to make it cross-browser compatible. Of all the other options available at the time, Flash had the biggest adoption and smallest download. I already had a copy of 3d Studio Max—so that was an obvious choice. Software for three-dimensional graphics is expensive and there still doesn't seem to be a good consumer-level option. PHP is a great and very well documented scripting language. I am not the best programmer so it is great to use a program with so many online examples to draw from. PHPMyAdmin is the only software I have ever really used for creating databases. I stumbled into this with no knowledge, read minimal documentation—and created a clunky database (I would structure it differently now) that works just fine. Of course PHP and MySQL and PHPMyAdmin are all open source projects that can be downloaded for free, which is how it should be!*

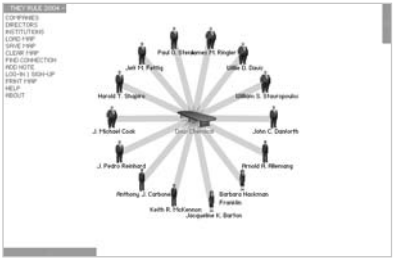
Why do you choose to work with software?

No other medium allows the same combination of massive persistent social collaboration and interaction—particularly software on the Internet where so many people (by no means all) can connect through software and create meaningful relationships with each other. We have only just begun to see the potential of the Internet for aiding social change. Unfortunately, the NSA has also discovered its potential for social control. It is a contested arena, and it is important that artists are amongst those in the fight!

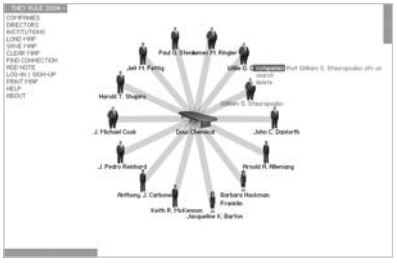
Notes

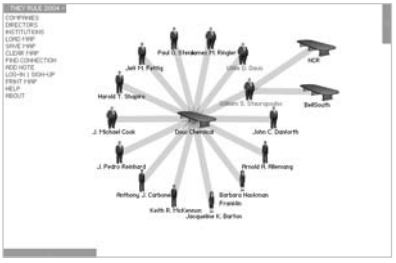
1. http://www.demos.org/inequality

Browse the list of companies and select one.

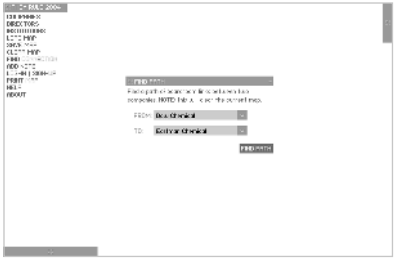


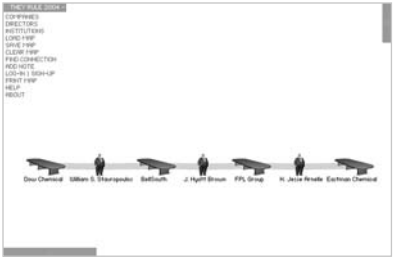
Reveal the board of directors.



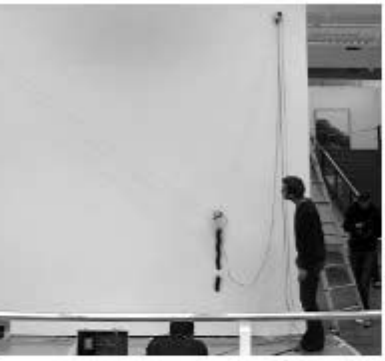


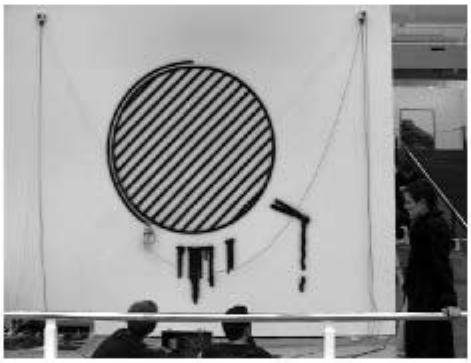
Select one board member and show the other boards on which he sits.

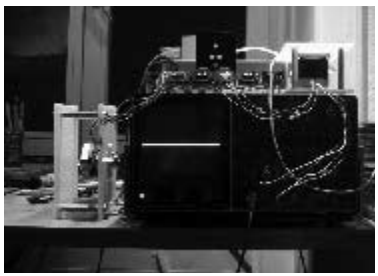




Search for and then display a path between two companies.







Hektor in its case (left) and undergoing testing (right).
Images courtesy of **Jürg Lehni and Uli Franke**.

Hektor and *Scriptographer* *(Interview with Jürg Lehni)*

Creators	Jürg Lehni and Uli Franke
Year	2001–2002
Medium	Custom Hardware, Plugin Software for Illustrator
Software	Scriptographer (C++, Java, JavaScript), PIC-Assembler
URL	www.hektor.ch, www.scriptographer.com, www.scratchdisk.com

What are *Hektor* and *Scriptographer*?

Hektor *is a portable spray-paint output device for laptop computers. It was created in close collaboration with the engineer Uli Franke for my diploma project at écal (école cantonale d'art de Lausanne) in 2002.* Hektor's *light and fragile installation consists only of two motors, toothed belts, and a can holder that handles regular spray cans. The can is moved along drawing paths, and during operation the mechanism sometimes trembles and wobbles and the paint often drips. The contrasts between these low-tech aspects and the high-tech touch of the construction hold ambiguous and poetic qualities and make* Hektor *enjoyable to watch in action.* Hektor *has been used for many projects in different contexts, often in collaboration with other designers and artists.*

Hektor *works with vector drawings and is controlled directly from Adobe Illustrator through the use of* Scriptographer. Scriptographer *is a scripting plug-in for Illustrator that is developed as open source software and is available under the GPL license. It is written in C++ and Java and exposes Illustrator's functionality to a Java virtual machine that is embedded in the application itself. It uses the Rhino JavaScript engine to execute scripts. This allows users with some knowledge of the JavaScript language to extend the functionality of Illustrator with their own definition of tools: mouse handlers, generative scripts, automated repeating tasks, etc. More advanced tools can also be written directly in Java, and Rhino's Java bridge can be used to interface with any Java library and the Java Core API directly from scripts. This leads to a great amount of possibilities, from serial port or network communication to database connections and advanced image manipulation, just to name a few.*

Why did you create *Hektor* and *Scriptographer*?

Hektor *was created with a certain attitude toward design and the use of tools. In the beginning there was an urge to go beyond the limitations of today's clean computer, screen, and vector graphic–based design. Intuition played an important role in the search for a new output device that would convey the abstract geometries contained in vector graphics in a different way than normal printers.*

The aim was to make a statement about design by providing a new tool for other designers and artists to experiment with, a tool with an inherently particular and distinctive aesthetic. Making the technology available to others and not only using it for my own purposes was an important step in the project. It was very interesting to see what people from different backgrounds were seeing in this machine and how they were working with the technical limitations and using them in the results rather than trying to hide them.

Today's desktop publishing design chain with all its standards and softwares has a strong influence on the aesthetics of the products. The tools offer predefined ways of working, and

"We Try Harder", 2002. Cornel Windlin and Hektor for the "Public Affairs" exhibition at the Kunsthaus Zurich, Switzerland. Images courtesy of Jürg Lehni and Uli Franke.

escaping these is not easy; it requires the user to be conscious of the limitations. Current software is mostly based on commonly known metaphors and simulation of real tools, but software is inherently different. It is modular and programmable and offers much more flexibility. Unfortunately the applications from most of the big companies still work the other way around—they are often predefined, inflexible, and monolithic.

The motivation for creating Scriptographer *and making it freely available was to provide a way to open up one of the main applications for graphic designers and to create a community around it that finds different approaches to graphic design by integrating programming in the workflow and the tools they already use.*

What software tools were used?

The circuit board for Hektor *was designed with Eagle. The controller software was written in PIC Assembler.* Scriptographer *was used to develop the algorithm for Hektor's damping movements to make sure the can does not start to tremble too much. A geometrical solution was found that adds tangential circles and tangents between them at places where too harsh movements would happen in the vector drawings.*

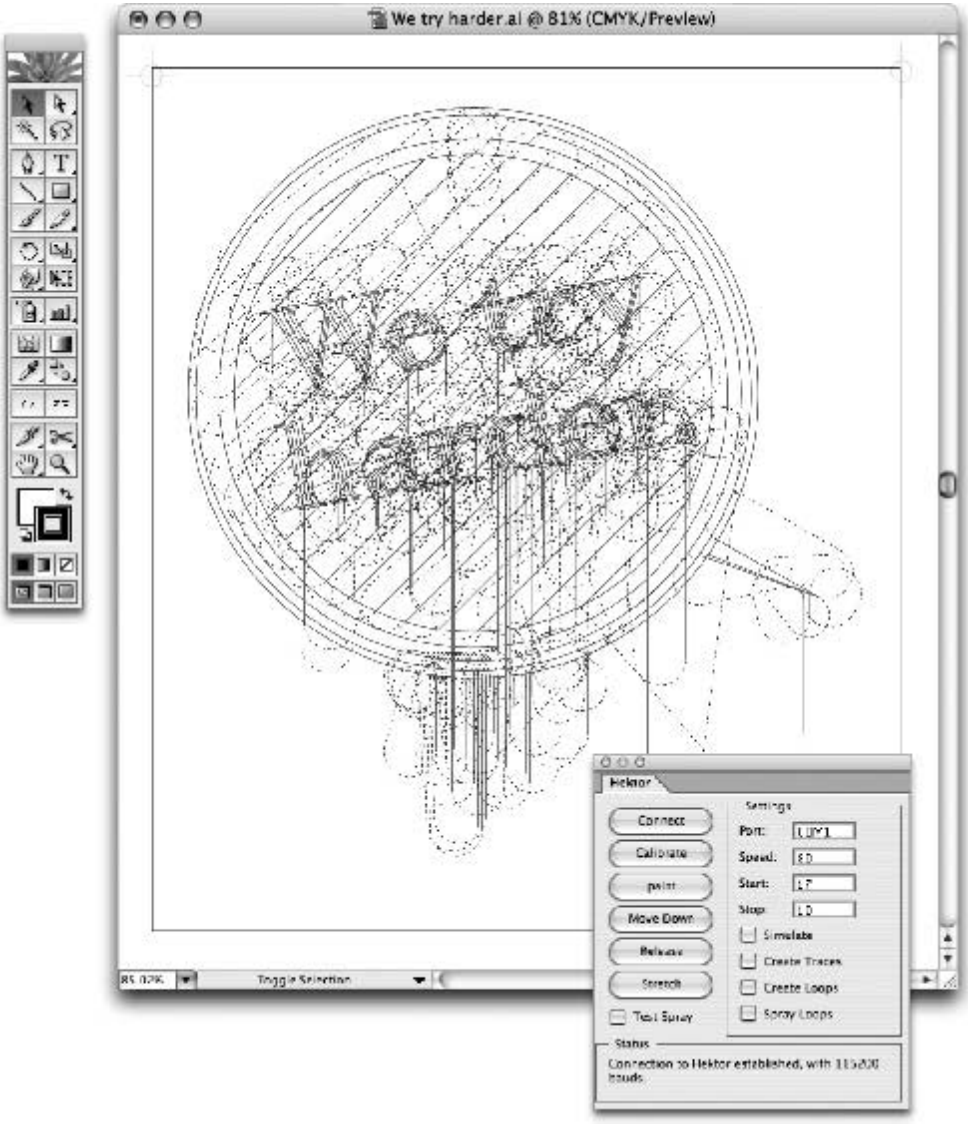
Scriptographer *also directly controls the stepper motors through the serial port interface and the PIC controller, to which it sends the movements for the drawings. In order for it to do so, the geometries must be converted from a Cartesian coordinate system into one based on triangulation.*

Why did you write your own software tools?

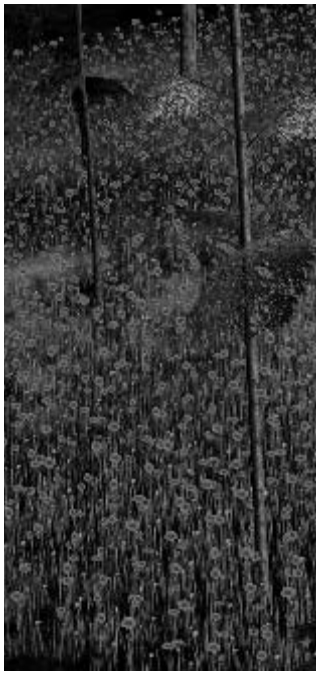
As already stated, there were certain ideological motivations for creating Hektor *and* Scriptographer. *But both projects are tools that are actually used, and both the ideological and the pragmatical parts of the projects are equally important. A year later I would have probably used Processing for the same task, but when work on* Scriptographer *started I was not aware of it, and the fact that* Scriptographer *nests itself in an existing application that plays an important role in graphic design helped underline the motivations of both projects.*

Why do you choose to work with software?

Computers fascinated me at an early age, when I started to tinker around with the Commodore VC-20 and later with my brother's C-64. This fascination was mostly based on the optical and audible outputs of these machines and has never really left me since. Over the years this led to learning many languages and concepts around computers. Programming is still my main tool of choice, which I constantly try to exploit and question in my work. I consider the ability to write programs (or better, formulate processes) as a freedom in the way computers are used, and I hope that it will become more and more common to work in such ways. Computers are the abstraction of tools that can simulate virtually any other tool. I believe that beyond the step of simulation there are many other possibilities to be discovered.



The *Scriptographer* plug-in for Adobe Illustrator by Jürg Lehni, running the Hektor software that computes motion paths and directly communicates with the hardware. Image courtesy of Jürg Lehni.



The Endless Forest *(Interview with Auriea Harvey & Michaël Samyn)*

Creators	Tale of Tales; Auriea Harvey and Michaël Samyn with Lina Kusaite, Ringtail Studios, Laura Smith, Jan Verschoren, Gerry De Mol, Ronald Jones
Year	2005+
Medium	Multiplayer Online Game
Software	Quest3D
URL	www.tale-of-tales.com/TheEndlessForest

What is *The Endless Forest*?

We call it a social screensaver—an online multiplayer game that runs as a screensaver on a computer. Every participant plays a deer in an eternal forest. The game does not contain violence, competition, or any chat functionality. It's all about playing a deer and interacting with other people in (gentle) deer ways. Additionally, The Endless Forest *is a stage for virtual performances in which the authors can create spectacles in real time. The total project is quite big. We are developing it step by step, in part inspired by the input of players. The first phase is relatively minimal in terms of size and interaction, but is nonetheless a pleasant experience.*

Why did you create *The Endless Forest*?

We think our contemporary world is a horrible place. Our cultures are being swallowed by the economic machine, politics have degraded into cheap television propaganda, and violence is condoned, if not encouraged, as the way of choice for dealing with any conflict. All of this is against a backdrop of ever-increasing poverty and the diluting of humanistic democratic ideals into free markets and globalization. Direct political action has been perverted by fashion and media and only contributes to the overwhelming climate of antagonism and violence. People are becoming increasingly sour, mean, and cynical—which throws us into a vicious circle that can only be broken by extreme events.

Rather than sitting around and waiting for this cataclysm, we are trying to break the spiral with our work. The Endless Forest *allows you to shake off all these worries momentarily and connect to something that you may have thought was lost. This something is partly inside of yourself, the place where you feel joy. But with its rich reference to cultural ideas (the forest, deer, religion, myth, etc.),* The Endless Forest *also connects you to more noble and beautiful aspects of our society that often get drowned in the "outside world." And last but not least, it is a social place: every other deer in the forest represents another human. You can interact with them in ways that other environments (virtual or otherwise) often don't allow.*

The Endless Forest *is a successor to an older project of ours,* Wirefire*. This was a real-time online performance tool made with Flash. For over three years, every Thursday night, the two of us would mix images, animations, and sounds in an improvised spectacle that anyone with a browser and an Internet connection could attend. While we kept getting better at creating spectacular shows, we began to feel the limitations of this system. First of all, for the environment to be truly alive, our presence was required. And second, the public had too little means to participate. The latter was a conscious choice. We do not believe that everyone can be creative at all times, and we believe that limiting the requirements for action from others allows us, the authors, to have more control and make better experiences.*

Still images from *The Endless Forest*. Images courtesy of the artists.

In multiuser environments, (some) people often end up ruining the experience of all. In The Endless Forest*, we take away the means by which they might be able to do such a thing. We limit communication to body language. The stories that get told are the ones you make up yourself based on the atmospheres we provide and interactions you have with others. The only things that you can do are nice things. So it gives you a vision of a perfectly harmonious society—an idea that players will, hopefully, take away from the game and perhaps (subconsciously) apply in their real lives.*

What software tools were used?

The game was programmed in Quest3D. The 3D modeling and animation was done in Blender and 3D Studio Max. The textures were made mostly with Photoshop.

Why did you use these tools?

Quest3D is a visual programming environment for real-time 3D. As such, it puts the power of programming into the hands of people who can use it in artistic ways. We think that one of the reasons why our interactions with computers can be so horrible and annoying is that software is made by one certain type of human, an engineer. An engineer is beyond any doubt a very creative person, but he (seldom she) tends to be more interested in systems and machines than in the humans that use them. Humans are the thing for which artists and designers know how to create. There are, however, very few tools that allow these artists and designers to create sophisticated interactive pieces. Quest3D is one of those rare tools.

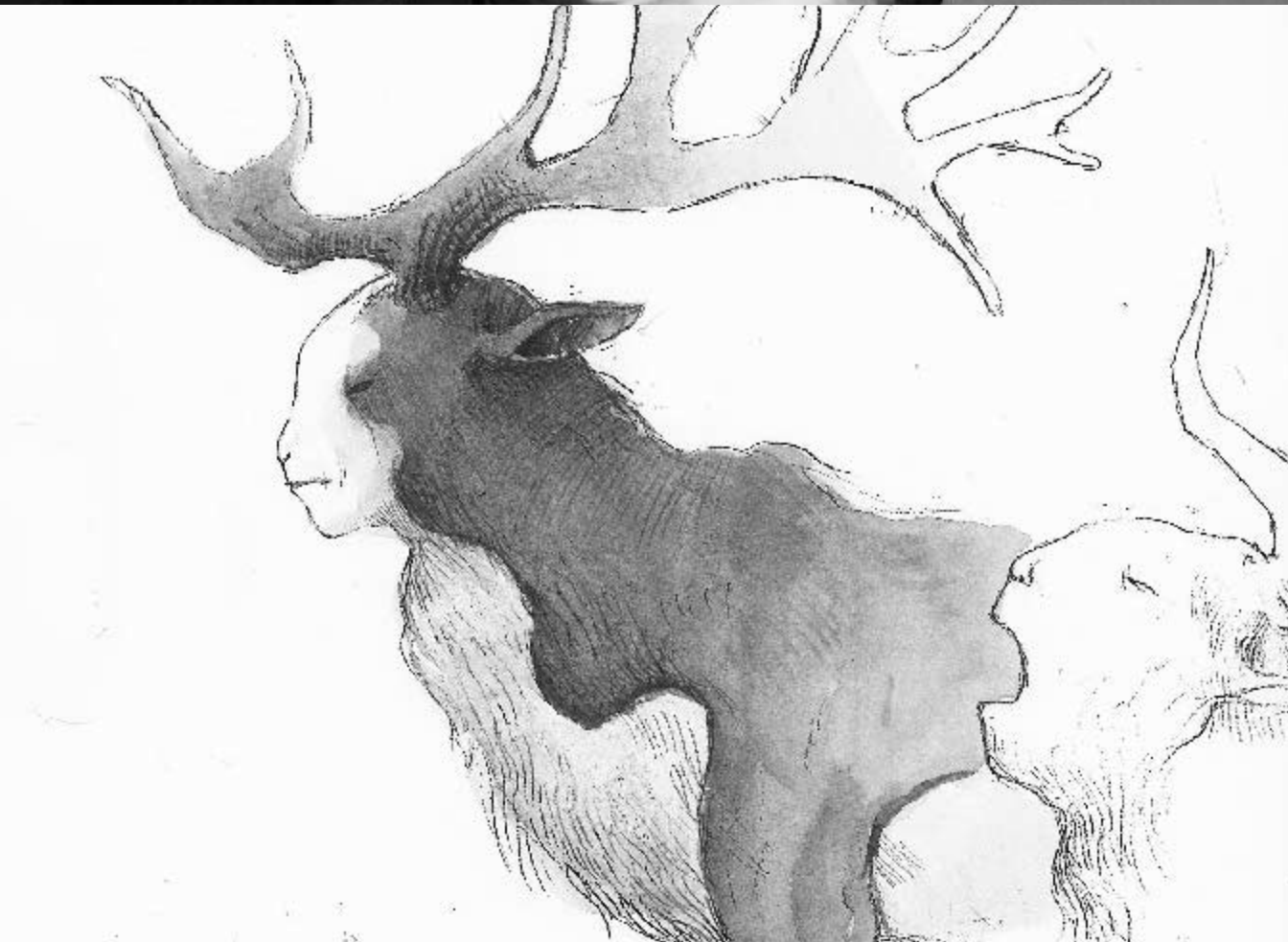
Also, as computers become more and more powerful, the range of things that you can make them do increases tremendously. If we continue to use the old tools (coding) to create computer applications, the only way to create these innovative projects is to continuously increase the number of engineers to work on it. New tools, like Quest3D, are required to deal with the ever-increasing potential of computers. These tools allow you to work on a higher, more abstract level, where you can have sufficient control over your content.

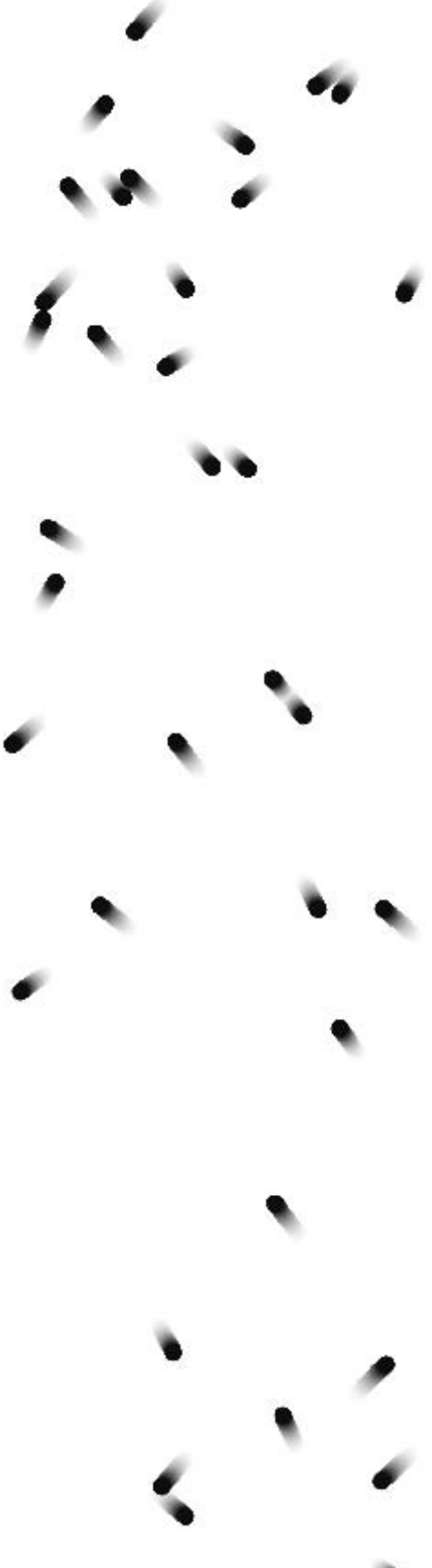
Why do you choose to work with software?

We have always been convinced that it is not required for new media art to have new media as its subject matter. Our work seeks to connect to a tradition of art that deals with humanity, cultures and symbols, myths and religions, emotions and aspirations, joy and humanity's tragedy. This does not, however, mean that our work is technophobic. On the contrary! We believe that we should attempt to exploit what is unique about computers and use it to our advantage. To simply create an image with a computer is wasteful when this machine has the potential for multisensory experiences, real-time processing, and (most of all) interaction. We see computer games as the most potent expression of the capabilities of these new media. But they are still very much stuck in their roots of games. We seek to emancipate game technology from these roots and use it to create forms of interactive art that are not necessarily about the rules and goals and rewards of games. This mission is motivated in part by our own frustrations with so many games. Sometimes they succeed so well in crafting believable environments and endearing characters, and then they end up ruining everything by throwing the game in your face and making you do things that you don't want to do (and that are often too hard to do, for us).

We want to take this technology away from the engineers and the accountants and start making interactive pieces that allow for a much more free-form experience where you can enjoy the environment without pressure to "do the game."

Concept drawing by Lina Kusaite (bottom) and high-resolution rendering (top). Images courtesy of the artists.





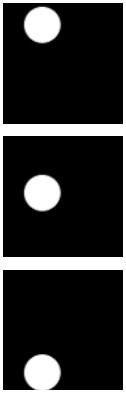
Motion 1: Lines, Curves

This unit introduces basic techniques for creating motion with code.

A deep understanding of motion and how to use it to communicate guides the art of the animator, dancer, and filmmaker. Practitioners of the media arts can employ motion in everything from dynamic websites to video games. The most fundamental component of motion is time, or more precisely, how elements change over time. Static images can communicate motion (think of Impressionist and Futurist paintings), and time-based media such as video, film, and software can express it directly. Defining motion through code displays the power and flexibility of the medium.

Controlling motion

To put a shape into motion, use a variable to change its attributes. We have already presented a few simple programs that move a shape across the screen (pp. 174–177). If clearing the screen before every frame is desired, remember to use the `background()` function at the beginning of the `draw()`. As mentioned in Structure 2 (p. 173), the `frameRate()` function may be used to control the number of frames drawn to the screen each second.



31-01

```
float y = 50.0;
float speed = 1.0;
float radius = 15.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
  ellipse(33, y, radius, radius);
  y = y + speed;
  if (y > height+radius) {
    y = -radius;
  }
}
```

You can create motion blur by drawing a transparent rectangle within `draw()`. This is an alternative to running `background()` at the beginning of `draw()` and can produce a subtle fade rather than a complete refresh. The effects of drawing this transparent rectangle slowly build each, frame by frame, to erase the previously drawn elements. The amount of blur is controlled by the transparency value used to draw the rectangle. Numbers closer to 255 will refresh the screen quickly, and numbers closer to 0 will create a longer fade.

Moving the images from top to bottom and then back again requires a variable that stores the direction of motion. In the following example, setting the `direction` variable to 1 moves the circle down and changing it to –1 moves the circle up. As the circle reaches the top and bottom of the screen, this variable changes by setting itself equal to its opposite value. So if `direction` is 1, then `-direction` will be -1. When `direction` is -1, the value of `-direction` is 1.

31-02

```
float y = 50.0;
float speed = 1.0;
float radius = 15.0;
int direction = 1;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  ellipse(33, y, radius, radius);
  y += speed * direction;
  if ((y > height-radius) || (y < radius)) {
    direction = -direction;
  }
}
```

To have the shape also change its position relative to the left and right edges of the display window requires a second set of variables for the x-coordinate. The following example uses the same principle as the previous one, but now defines the position of the shape on both the x-axis and y-axis. The shape reverses its direction when it reaches the edge.

31-03

```
float x = 50.0;       // X-coordinate
float y = 50.0;       // Y-coordinate
float radius = 15.0;  // Radius of the circle
float speedX = 1.0;   // Speed of motion on the x-axis
float speedY = 0.4;   // Speed of motion on the y-axis
int directionX = 1;   // Direction of motion on the x-axis
int directionY = -1;  // Direction of motion on the y-axis

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  ellipse(x, y, radius, radius);
  x += speedX * directionX;
  if ((x > width-radius) || (x < radius)) {
    directionX = -directionX;  // Change direction
  }
  y += speedY * directionY;
  if ((y > height-radius) || (y < radius)) {
    directionY = -directionY;  // Change direction
  }
}
```

It's possible to change not only the position of a shape but also its background value, stroke and fill values, and size. This example changes the size of four ellipses using the same technique for reversing direction explained in the previous two examples.

31-04

```
float d = 20.0;
float speed = 1.0;
int direction = 1;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  fill(255, 204);
}
```

31-04 cont.

```
void draw() {
  background(0);
  ellipse(0, 50, d, d);
  ellipse(100, 50, d, d);
  ellipse(50, 0, d, d);
  ellipse(50, 100, d, d);
  d += speed * direction;
  if ((d > width) || (d < width/10)) {
    direction = -direction;
  }
}
```

In contrast to the implicit motion presented in the previous examples, shapes can explicitly move from one position to another. This type of motion requires setting a start position and a distance to travel, and then moving between them over a series of frames. In the code below, the `beginX` and `beginY` variables represent the starting position. The `endX` and `endY` variables denote the final position. The `x` and `y` variables are the current position. The `step` variable determines what percentage of the total distance to travel each frame, and the `pct` variable keeps track of the total percentage of the distance traveled. The values of the `step` and `pct` variables must always be between 0.0 and 1.0. As the program runs, the `step` increases to change the position of the `x` and `y` variables, and when the percentage is no longer less than 1.0, the motion stops.

31-05

```
float beginX = 20.0;  // Initial x-coordinate
float beginY = 10.0;  // Initial y-coordinate
float endX = 70.0;    // Final x-coordinate
float endY = 80.0;    // Final y-coordinate
float distX;          // X-axis distance to move
float distY;          // Y-axis distance to move
float x = 0.0;        // Current x-coordinate
float y = 0.0;        // Current y-coordinate
float step = 0.02;    // Size of each step (0.0 to 1.0)
float pct = 0.0;      // Percentage traveled (0.0 to 1.0)

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  distX = endX - beginX;
  distY = endY - beginY;
}

void draw() {
  fill(0, 12);
```

```
  rect(0, 0, width, height);
  pct += step;
  if (pct < 1.0) {
    x = beginX + (pct * distX);
    y = beginY + (pct * distY);
  }
  fill(255);
  ellipse(x, y, 20, 20);
}
```

31-05 cont.

The easing technique introduced in Input 4 (p. 237) can be used to slow the shape down as it reaches its target position. The following example adapts the code from the previous example to make the shape decelerate as it approaches its destination.

31-06

```
float x = 20.0;        // Initial x-coordinate
float y = 10.0;        // Initial y-coordinate
float targetX = 70.0;  // Destination x-coordinate
float targetY = 80.0;  // Destination y-coordinate
float easing = 0.05;   // Size of each step along the path

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  float d = dist(x, y, targetX, targetY);
  if (d > 1.0) {
    x += (targetX - x) * easing;
    y += (targetY - y) * easing;
  }
  fill(255);
  ellipse(x, y, 20, 20);
}
```

Moving along curves

The simple curves explained in Math 2 (p. 79) can provide paths for shapes in motion. Instead of drawing the entire curve in one frame, it's possible to calculate each step of the curve on consecutive frames. The following example presents a very general way to write this code. Changing the variables at the top of the code changes the start and stop position, the curve shape, and the number of steps to take along the curve.

31-07

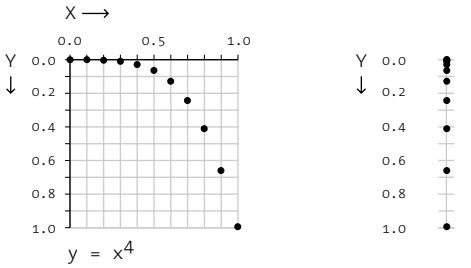
```
float beginX = 20.0;   // Initial x-coordinate
float beginY = 10.0;   // Initial y-coordinate
float endX = 70.0;     // Final x-coordinate
float endY = 80.0;     // Final y-coordinate
float distX;           // X-axis distance to move
float distY;           // Y-axis distance to move
float exponent = 0.5;  // Determines the curve
float x = 0.0;         // Current x-coordinate
float y = 0.0;         // Current y-coordinate
float step = 0.01;     // Size of each step along the path
float pct = 0.0;       // Percentage traveled (0.0 to 1.0)

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  distX = endX - beginX;
  distY = endY - beginY;
}

void draw() {
  fill(0, 2);
  rect(0, 0, width, height);
  pct += step;
  if (pct < 1.0) {
    x = beginX + (pct * distX);
    y = beginY + (pow(pct, exponent) * distY);
  }
  fill(255);
  ellipse(x, y, 20, 20);
}
```

All of the basic curves presented on page 82 can be scaled and combined to generate unique paths of motion. Once a step along one curve has been calculated, the program can switch to calculating positions based on a different curve.

31-08

```
float beginX = 20.0;   // Initial x-coordinate
float beginY = 10.0;   // Initial y-coordinate
float endX = 70.0;     // Final x-coordinate
float endY = 80.0;     // Final y-coordinate
float distX;           // X-axis distance to move
float distY;           // Y-axis distance to move
float exponent = 3.0;  // Determines the curve
float x = 0.0;         // Current x-coordinate
float y = 0.0;         // Current y-coordinate
float step = 0.01;     // Size of each step along the path
float pct = 0.0;       // Percentage traveled (0.0 to 1.0)
int direction = 1;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  distX = endX - beginX;
  distY = endY - beginY;
}

void draw() {
  fill(0, 2);
  rect(0, 0, width, height);
  pct += step * direction;
  if ((pct > 1.0) || (pct < 0.0)) {
    direction = direction * -1;
  }
  if (direction == 1) {
    x = beginX + (pct * distX);
    float e = pow(pct, exponent);
    y = beginY + (e * distY);
  } else {
    x = beginX + (pct * distX);
    float e = pow(1.0-pct, exponent*2);
    y = beginY + (e * -distY) + distY;
  }
  fill(255);
  ellipse(x, y, 20, 20);
}
```

As a shape moves along a curve, its speed changes. A curve can be used to control the speed of a visual element that moves on a straight line. The distance between the points can be plotted along each dimension to show the distance between each step. When the curve is mostly horizontal (flat), there is very little vertical distance, but as the curves become more vertical (steep), the distance along the y-axis increases exponentially. The images below show how when the x values in the equation $y = x^4$ are increased at a constant rate, the y values grow exponentially. The image on the left displays the y values corresponding to x values from 0.0 to 1.0 at increments of 0.1. The image on the right displays only the y value without the context of the x value to reveal the increasing changes:



The following example shows changing the speed of a shape using the curve in the above image. The circle begins very slowly, gradually accelerates, and then stops at the bottom of the display window. The `exponent` variable defines the slope of the curve, which changes the rate of motion. Click the mouse to select a new starting point.

31-09

```
float beginX = 20.0;   // Initial x-coordinate
float beginY = 10.0;   // Initial y-coordinate
float endX = 70.0;     // Final x-coordinate
float endY = 80.0;     // Final y-coordinate
float distX;           // X-axis distance to move
float distY;           // Y-axis distance to move
float exponent = 3.0;  // Determines the curve
float x = 0.0;         // Current x-coordinate
float y = 0.0;         // Current y-coordinate
float step = 0.01;     // Size of each step along the path
float pct = 0.0;       // Percentage traveled (0.0 to 1.0)

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  distX = endX - beginX;
  distY = endY - beginY;
}
```

31-09 cont.

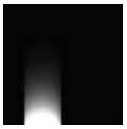
```
void draw() {
  fill(0, 2);
  rect(0, 0, width, height);
  if (pct < 1.0) {
    pct = pct + step;
    float rate = pow(pct, exponent);
    x = beginX + (rate * distX);
    y = beginY + (rate * distY);
  }
  fill(255);
  ellipse(x, y, 20, 20);
}

void mousePressed() {
  pct = 0.0;
  beginX = x;
  beginY = y;
  distX = mouseX - x;
  distY = mouseY - y;
}
```

Motion through transformation

The transformation functions can also create motion by changing the parameters to `translate()`, `rotate()`, and `scale()`. Before using transformations for motion, it's important to acknowledge that transformations reset at the beginning of each `draw()`. Therefore, running `translate(5,0)` will always move the coordinate system 5 pixels to the right in each frame. It will not move the system 5 right on the first frame, 10 on the next, 15 on the next etc. In the same way, translations inside `setup()` have no effect on the shapes rendered in `draw()`.

31-10

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
  translate(50, 0);  // Has no effect
}

void draw() {
  background(0);
  ellipse(0, 50, 60, 60);
}
```



31-11

```
float y = 50.0;
float speed = 1.0;
float radius = 15.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  translate(0, y);  // Set the y-coordinate of the circle
  ellipse(33, 0, radius, radius);
  y += speed;
  if (y > height+radius) {
    y = -radius;
  }
}
```

31-12

```
float y = 50.0;
float speed = 1.0;
float radius = 15.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  pushMatrix();
  translate(0, y);
  // Affected by first translate()
  ellipse(33, 0, radius, radius);
  translate(0, y);
```

```
  // Affected by first and second translate()
  ellipse(66, 0, radius, radius);
  popMatrix();
  // Not affected by either translate()
  ellipse(99, 50, radius, radius);
  y = y + speed;
  if (y > height+radius) {
    y = -radius;
  }
}
```

31-12 cont.

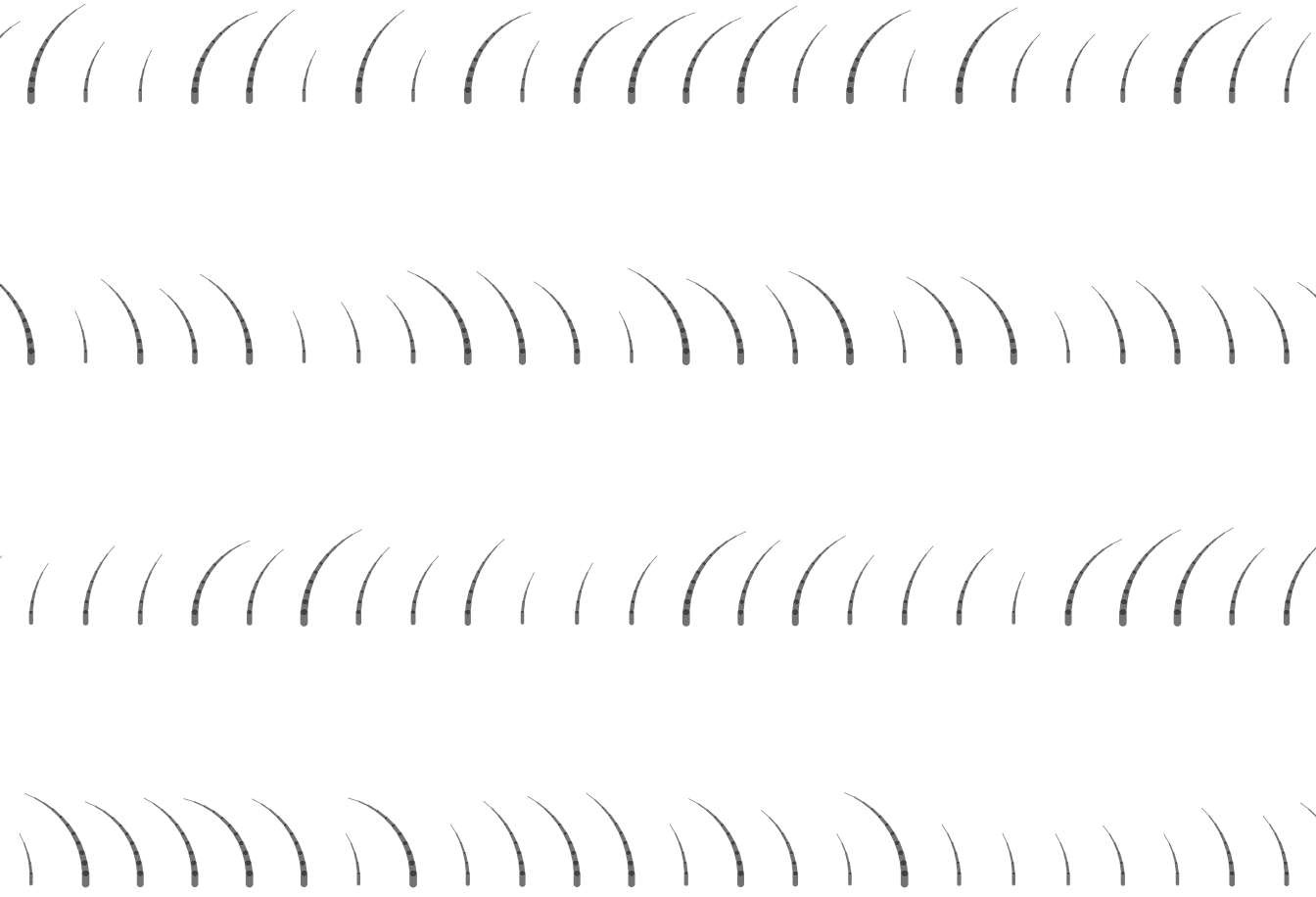
```
float angle = 0.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  angle = angle + 0.02;
  translate(70, 40);
  rotate(angle);
  rect(-30, -30, 60, 60);
}
```

31-13

The transformation functions `translate()`, `rotate()`, and `scale()` can be used to generate motion, but using them together can be tricky. This is discussed in detail in Transform 2 (p. 137).

Exercises

1. *Move two shapes continuously, but constrain their positions to the display window.*
2. *Move three shapes on different curves to create a kinetic composition.*
3. *Use the transformation functions to animate a shape.*



Motion 2: Machine, Organism

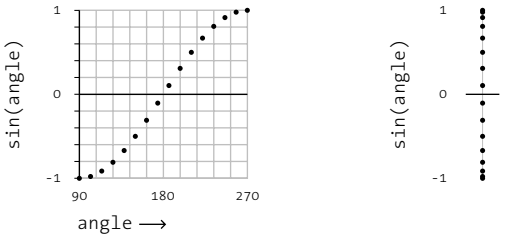
This unit introduces the communicative aspects of motion—how different qualities of motion create diverse moods and meanings.

Programmers determine how elements in their programs will move. These decisions influence how the forms are interpreted by viewers. Over the course of evolution, humans have developed instincts and attitudes toward the motion in the world around us. We classify and categorize elements in the world by how they move or do not move. We make distinctions between animate and inanimate things. Motion in software can take advantage of humans' innate understanding of movement or ignore it. Software makes possible countless types of motion, but two major categories of interest are mechanical and organic motion. This unit focuses on the qualities that define these types of motion.

Mechanical motion

In the catalog for The Museum of Modern Art's 1934 Machine Art exhibition, Alfred Barr described characteristics of the machines of that time: "Machines are, visually speaking, a practical application of geometry. Forces which act in straight lines are changed in direction and degree by machines which are themselves formed of straight lines and curves. The lever is geometrically a straight line resting on a point. The wheel and axle is composed of concentric circles and radiating straight lines." He further explained that motion "increases their aesthetic interest, principally through the addition of temporal rhythms." While machines and society's ideas about machines have changed dramatically in the last seventy years, Barr's insights remain relevant in defining the characteristics of mechanical motion. Prototypical examples of machine motion include the clock, the metronome, and the piston. These mechanisms' movements are all characterized by regular rhythm, repetition, and efficiency. Writing code to produce these qualities of motion communicates the essence of the machines through software.

The `sin()` function is often used to produce elegant motion. It can generate an accelerating and decelerating speed as a shape moves from one frame to another. The images below show how the speed along a sine wave is consistent if the angle is increased at a constant rate, but the speed along the y-axis increases and decreases at the top and bottom of the curve. The image on the left displays the changing sine value as the angle grows. The image on the right displays only the sine values but does not draw the x-axis, to emphasize the changes in speed along the y-axis:



The values from `sin()` are used to create the motion for a shape in the following example. The `angle` variable increases continually to produce changing values from `sin()` in the range of –1 to 1. These values are multiplied by the `radius` variable to magnify the values. The result is assigned to the `yoffset` variable and is then used to determine the y-coordinate of the ellipse on the following line. Notice how the circle slows down at the top and bottom of the screen and accelerates in the middle.

32-01

```
float angle = 0.0;    // Current angle
float speed = 0.1;    // Speed of motion
float radius = 40.0;  // Range of motion

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  angle += speed;
  float sinval = sin(angle);
  float yoffset = sinval * radius;
  ellipse(50, 50 + yoffset, 80, 80);
}
```

Adding values from `sin()` and `cos()` can produce more complex movement that remains periodic. In this example, a small dot moves in a circular pattern using values from `sin()` and `cos()`. A larger dot uses the same values for its base position but adds additional `sin()` and `cos()` calculations to produce an offset. You can easily see the difference between the two movements by looking at the positions of each point as the program runs.

sx=1.0
sy=0.5

sx=1.0
sy=1.5

sx=1.0
sy=2.0

sx=1.0
sy=3.0

sx=1.5
sy=1.0

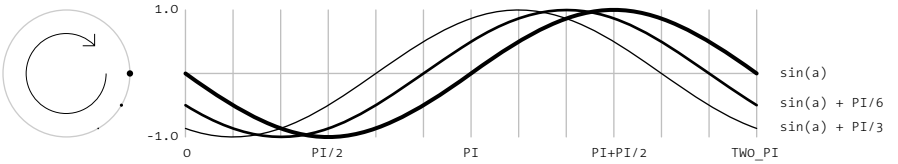
sx=2.0
sy=1.0

sx=3.0
sy=1.0

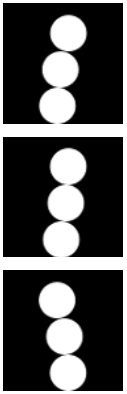
```
float angle = 0.0;    // Current angle
float speed = 0.05;   // Speed of motion
float radius = 30.0;  // Range of motion
float sx = 2.0;
float sy = 2.0;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  fill(0, 4);
  rect(0, 0, width, height);
  angle += speed;  // Update the angle
  float sinval = sin(angle);
  float cosval = cos(angle);
  // Set the position of the small circle based on new
  // values from sine and cosine
  float x = 50 + (cosval * radius);
  float y = 50 + (sinval * radius);
  fill(255);
  ellipse(x, y, 2, 2);  // Draw smaller circle
  // Set the position of the large circles based on the
  // new position of the small circle
  float x2 = x + cos(angle * sx) * radius/2;
  float y2 = y + sin(angle * sy) * radius/2;
  ellipse(x2, y2, 6, 6);  // Draw larger circle
}
```

The *phase* of a function is one iteration through its possible values—for example, a single rise and fall sequence of a sine curve. *Phase shifting* occurs when the function is offset to start at a different point within the phase, such as the downward part of a sine curve rather than the top. Musical, visual, and numeric sequences all have phases, as do physical phenomena such as light waves. In the sound domain, Steve Reich's *Piano Phases* is a composition in which two pianos play the same sequence of notes, but one musician plays faster, creating a continuous phase shift as notes move in and out of synchrony. The player with the faster tempo periodically moves back into phase with the slower player, but is first offset by one note, then two notes, then three, etc. The performance ends when both pianos are back in phase and are playing the notes at the same time. In the visual domain, phase shifting creates the complex moire patterns that result when two identical patterns are superimposed and shifted. In trigonometry, a cosine wave is a sine wave offset by 90° (p. 121). Shifting the angle used to generate

values from `sin()` provides the same sequence of numbers, but shifted across frames of animation:



The following two examples change the x-coordinate and diameter of circles to demonstrate phase shifting. In the first example, each circle has the same horizontal motion, but the motion is offset in time. In the second example, each circle has the same position and rate of growth cycle, but the growth cycle is offset.



32-03

```
float angle = 0.0;
float speed = 0.1;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  background(0);
  angle = angle + speed;
  ellipse(50 + (sin(angle + PI) * 5), 25, 30, 30);
  ellipse(50 + (sin(angle + HALF_PI) * 5), 55, 30, 30);
  ellipse(50 + (sin(angle + QUARTER_PI) * 5), 85, 30, 30);
}
```

32-04

```
float angle = 0.0;   // Changing angle
float speed = 0.05;  // Speed of growth

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  fill(255, 180);
}

void draw() {
  background(0);
```

32-04
cont.

```
  circlePhase(0.0);
  circlePhase(QUARTER_PI);
  circlePhase(HALF_PI);
  angle += speed;
}

void circlePhase(float phase) {
  float diameter = 65 + (sin(angle + phase) * 45);
  ellipse(50, 50, diameter, diameter);
}
```

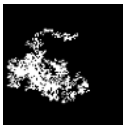
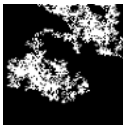
Organic motion

Examples of organic movement include a leaf falling, an insect walking, a bird flying, a person breathing, a river flowing, and smoke rising. This type of motion is often considered idiosyncratic and stochastic.

Explorations in photography have led to a new understanding of human and animal motion. Étienne-Jules Marey and Eadweard Muybridge focused the lenses of their cameras on bodies in motion. In the 1880s Marey famously captured birds in flight, revealing the changing shapes of wings in image montages. Muybridge's sensational photographs of horses in motion used fifty cameras in sequence along a track to capture still images of a running horse. These photographs provided visual evidence regarding the horse's stride that had previously been impossible to collect. In the 1930s, Harold Edgerton invented technologies for capturing unique movements such as the beating of a hummingbird's wings and the motion of a starfish across the sea floor. Such images and films have changed the way the world is understood and can inform the way organic motion is approached using software.

Software explorations within the last twenty years have also provided a new understanding of organic motion. The Boids (pp. 473–475) software created by Craig Reynolds in 1986 simulates the flocking behavior found in birds and fish and has led to a new understanding of these emergent behaviors of animals. Karl Sims's Evolved Virtual Creatures from 1994 presents an artificial evolution, focusing on locomotion, where virtual block creatures engage in walking, jumping, and swimming competitions. The simple blocks demonstrate extraordinary emotive qualities as they twist and turn, appearing to struggle in their pursuit of locomotion.

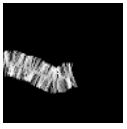
Brownian motion, named in honor of the botanist Robert Brown, is jittery, stochastic motion that was originally ascribed to the movements of minute particles within fluids or the air; it appears entirely random. This motion can be simulated in software by setting a new position for a particle each frame, without preference as to the direction of motion. Leaving a trail of the previous positions of an element is a good technique for tracing its path through space.

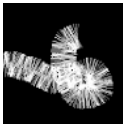



32-05

```
float x = 50.0;        // X-coordinate
float y = 80.0;        // Y-coordinate

void setup() {
  size(100, 100);
  randomSeed(0);       // Force the same random values
  background(0);
  stroke(255);
}

void draw() {
  x += random(-2, 2);  // Assign new x-coordinate
  y += random(-2, 2);  // Assign new y-coordinate
  point(x, y);
}
```

The `sin()` and `cos()` functions can be used to create unpredictable motion when employed with the `random()` function. The following example presents a line with a position and direction, and every frame the direction is changed by a small value between –0.3 and 0.3. The position of the line at each frame is based on its current position and the slight variation to its direction. The `cos()` function uses the angle to calculate the next x-coordinate for the line, and the `sin()` function uses the same angle to calculate the next y-coordinate.

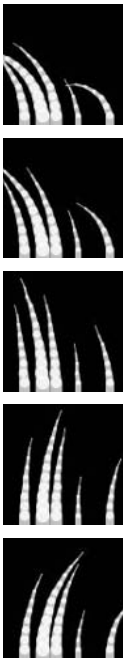




32-06

```
float x = 0.0;       // X-coordinate
float y = 50.0;      // Y-coordinate
float angle = 0.0;   // Direction of motion
float speed = 0.5;   // Speed of motion

void setup() {
  size(100, 100);
  background(0);
  stroke(255, 130);
  randomSeed(121);   // Force the same random values
}

void draw() {
  angle += random(-0.3, 0.3);
  x += cos(angle) * speed;  // Update x-coordinate
  y += sin(angle) * speed;  // Update y-coordinate
  translate(x, y);
  rotate(angle);
  line(0, -10, 0, 10);
}
```

The following example is an animated extension of code 22-04 (p. 200). Here the `angle` variable for the `tail()` function is continually changing to produce a swaying motion. Because the angles for each shape accumulate with each unit, the longest shapes with the most units swing from side to side with a greater curvature.



32-07

```
float inc = 0.0;

void setup() {
  size(100, 100);
  stroke(255, 204);
  smooth();
}

void draw() {
  background(0);
  inc += 0.01;
  float angle = sin(inc)/10.0 + sin(inc*1.2)/20.0;
  tail(18, 9, angle/1.3);
  tail(33, 12, angle);
  tail(44, 10, angle/1.3);
  tail(62, 5, angle);
  tail(88, 7, angle*2);
}

void tail(int x, int units, float angle) {
  pushMatrix();
  translate(x, 100);
  for (int i = units; i > 0; i--) {
    strokeWeight(i);
    line(0, 0, 0, -8);
    translate(0, -8);
    rotate(angle);
  }
  popMatrix();
}
```

The `noise()` function introduced in Math 4 (p. 127) is another resource for producing organic motion. Because the numbers returned from `noise()` are easy to control, they are a good way to add subtle irregularity to movement. The following example draws two lines to the screen and sets their endpoints based on numbers returned from `noise()`.

32-08

```
float inc1 = 0.1;
float n1 = 0.0;
float inc2 = 0.09;
float n2 = 0.0;

void setup() {
  size(100, 100);
  stroke(255);
  strokeWeight(20);
  smooth();
}

void draw() {
  background(0);
  float y1 = (noise(n1) - 0.5) * 30.0; // Values -15 to 15
  float y2 = (noise(n2) - 0.5) * 30.0; // Values -15 to 15
  line(0, 50, 40, 50 + y1);
  line(100, 50, 60, 50 + y2);
  n1 += inc1;
  n2 += inc2;
}
```

The `noise()` function can also be used to generate dynamic textures. In the following example, the first two parameters are used to produce a two-dimensional texture and the third parameter increases its value each frame to vary the texture. Changing the `density` parameter increases the image resolution, and changing the `inc` parameter changes the texture resolution.

32-09

```
float inc = 0.06;
int density = 4;
float znoise = 0.0;

void setup() {
  size(100, 100);
  noStroke();
}

void draw() {
  float xnoise = 0.0;
  float ynoise = 0.0;
  for (int y = 0; y < height; y += density) {
    for (int x = 0; x < width; x += density) {
      float n = noise(xnoise, ynoise, znoise) * 256;
      fill(n);
```

```
        rect(y, x, density, density);
        xnoise += inc;
      }
      xnoise = 0;
      ynoise += inc;
    }
    znoise += inc;
  }
```

32-09
cont.

Exercises

1. *Make a shape move with numbers returned from* `sin()` *and* `cos()`.
2. *Develop a kinetic composition using the concept of phase shifting.*
3. *Use code 32-06 as a base for creating a more advanced organism.*

Data 4: Arrays

This unit introduces arrays of data.

Syntax introduced:
`Array, [] (array access), new, Array.length`
`append(), shorten(), expand(), arraycopy()`

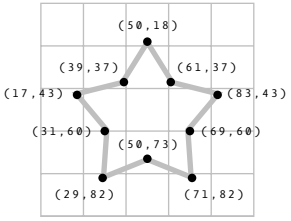
The term array refers to a structured grouping or an imposing number—"The dinner buffet offers an array of choices," "The city of Los Angeles faces an array of problems." In computer programming, an array is a set of data elements stored under the same name. Arrays can be created to hold any type of data, and each element can be individually assigned and read. There can be arrays of numbers, characters, sentences, boolean values, etc. Arrays might store vertex data for complex shapes, recent keystrokes from the keyboard, or data read from a file.

Five integer variables (1919, 1940, 1975, 1976, 1990) can be stored in one integer array rather than defining five separate variables. For example, let's call this array "dates" and store the values in sequence:

dates	1919	1940	1975	1976	1990
	[0]	[1]	[2]	[3]	[4]

Array elements are numbered starting with zero, which may seem confusing at first but is important for more advanced programming. The first element is at position [0], the second is at [1], and so on. The position of each element is determined by its offset from the start of the array. The first element is at position [0] because it has no offset; the second element is at position [1] because it is offset one place from the beginning. The last position in the array is calculated by subtracting 1 from the array length. In this example, the last element is at position [4] because there are five elements in the array.

Arrays can make the task of programming much easier. While it's not necessary to use them, they can be valuable structures for managing data. Let's begin with a set of data points we want to add to our program in order to draw a star:



The following example to draw this shape demonstrates some of the benefits of using arrays, like avoiding the cumbersome chore of storing data points in individual

variables. The star has 10 vertex points, each with 2 values, for a total of 20 data elements. Inputting this data into a program requires either creating 20 variables or using an array. The code (below) on the left demonstrates using separate variables. The code in the middle uses 10 arrays, one for each point of the shape. This use of arrays improves the situation, but we can do better. The code on the right shows how the data elements can be logically grouped together in two arrays, one for the x-coordinates and one for the y-coordinates.

Separate variables

```
int x0 = 50;
int y0 = 18;
int x1 = 61;
int y1 = 37;
int x2 = 83;
int y2 = 43;
int x3 = 69;
int y3 = 60;
int x4 = 71;
int y4 = 82;
int x5 = 50;
int y5 = 73;
int x6 = 29;
int y6 = 82;
int x7 = 31;
int y7 = 60;
int x8 = 17;
int y8 = 43;
int x9 = 39;
int y9 = 37;
```

One array for each point

```
int[] p0 = { 50, 18 };
int[] p1 = { 61, 37 };
int[] p2 = { 83, 43 };
int[] p3 = { 69, 60 };
int[] p4 = { 71, 82 };
int[] p5 = { 50, 73 };
int[] p6 = { 29, 82 };
int[] p7 = { 31, 60 };
int[] p8 = { 17, 43 };
int[] p9 = { 39, 37 };
```

One array for each axis

```
int[] x = { 50, 61, 83, 69, 71,
            50, 29, 31, 17, 39 };
int[] y = { 18, 37, 43, 60, 82,
            73, 82, 60, 43, 37 };
```

This example shows how to use the arrays within a program. The data for each vertex is accessed in sequence with a `for` structure. The syntax and usage of arrays is discussed in more detail in the following pages.

```
int[] x = { 50, 61, 83, 69, 71, 50, 29, 31, 17, 39 };
int[] y = { 18, 37, 43, 60, 82, 73, 82, 60, 43, 37 };

beginShape();
// Reads one array element every time through the for()
for (int i = 0; i < x.length; i++) {
  vertex(x[i], y[i]);
}
endShape(CLOSE);
```

33-01

Using arrays

Arrays are declared similarly to other data types, but they are distinguished with brackets, [and]. When an array is declared, the type of data it stores must be specified. After the array is declared, the array must be created with the keyword "new." This additional step allocates space in the computer's memory to store the array's data. After the array is created, the values can be assigned. There are different ways to declare, create, and assign arrays. In the following examples explaining these differences, an array with five elements is created and filled with the values 19, 40, 75, 76, and 90. Note the different way each method for creating and assigning elements of the array relates to `setup()`.

33-02

```
int[] data;               // Declare
void setup() {
  size(100, 100);
  data = new int[5];      // Create
  data[0] = 19;           // Assign
  data[1] = 40;
  data[2] = 75;
  data[3] = 76;
  data[4] = 90;
}
```

33-03

```
int[] data = new int[5];  // Declare, create
void setup() {
  size(100, 100);
  data[0] = 19;           // Assign
  data[1] = 40;
  data[2] = 75;
  data[3] = 76;
  data[4] = 90;
}
```

33-04

```
int[] data = { 19, 40, 75, 76, 90 };  // Declare, create, assign
void setup() {
  size(100, 100);
}
```

The previous three examples assume the arrays are used in a sketch with `setup()` and `draw()`. If arrays are not used with these functions, they can be created and assigned in the simpler ways shown in the following examples.

```
int[] data;         // Declare                                    33-05
data = new int[5];  // Create
data[0] = 19;       // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```

```
int[] data = new int[5];  // Declare, create                      33-06
data[0] = 19;             // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```

```
int[] data = { 19, 40, 75, 76, 90 };  // Declare, create, assign  33-07
```

The declare, create, and assign steps allow an array's values to be read. An array element is accessed using the name of the variable followed by brackets around the position from which you are trying to read.

```
int[] data = { 19, 40, 75, 76, 90 };                             33-08
line(data[0], 0, data[0], 100);
line(data[1], 0, data[1], 100);
line(data[2], 0, data[2], 100);
line(data[3], 0, data[3], 100);
line(data[4], 0, data[4], 100);
```

Remember, the first element in the array is in the 0 position. If you try to access a member of the array that lies outside the array boundaries, your program will terminate and give an `ArrayIndexOutOfBoundsException`.

```
int[] data = { 19, 40, 75, 76, 90 };                             33-09
println(data[0]);  // Prints 19 to the console
println(data[2]);  // Prints 75 to the console
println(data[5]);  // ERROR! The last element of the array is 4
```

The `length` field stores the number of elements in an array. This field is stored within the array and can be accessed with the dot operator (p. 107–108). The following example demonstrates how to utilize it.

```
int[] data1 = { 19, 40, 75, 76, 90 };
int[] data2 = { 19, 40 };
int[] data3 = new int[127];
println(data1.length); // Prints "5" to the console
println(data2.length); // Prints "2" to the console
println(data3.length); // Prints "127" to the console
```
33-10

Usually, a `for` structure is used to access array elements, especially with large arrays. The following example draws the same lines as code 33-08 but uses a `for` structure to iterate through every value in the array.

```
int[] data = { 19, 40, 75, 76, 90 };
for (int i = 0; i < data.length; i++) {
  line(data[i], 0, data[i], 100);
}
```
33-11

A `for` structure can also be used to put data inside an array—for instance, it can calculate a series of numbers and then assign each value to an array element. The following example stores the values from the `sin()` function in an array within `setup()` and then displays these values as the stroke values for lines within `draw()`.

```
float[] sineWave = new float[width];

for (int i = 0; i < width; i++) {
  // Fill the array with values from sin()
  float r = map(i, 0, width, 0, TWO_PI);
  sineWave[i] = abs(sin(r));
}

for (int i = 0; i < sineWave.length; i++) {
  // Set the stroke values to numbers read from the array
  stroke(sineWave[i] * 255);
  line(i, 0, i, height);
}
```
33-12

Storing the coordinates of many elements is another way to use arrays to make a program easier to read and manage. In the following example, the `x[]` array stores the x-coordinate for each of the 12 elements in the array, and the `speed[]` array stores a rate corresponding to each. Writing this program without arrays would have required 24 separate variables. Instead, it's written in a flexible way; simply changing the value assigned to `numLines` sets the number of elements drawn to the screen.

33-13

```
int numLines = 12;
float[] x = new float[numLines];
float[] speed = new float[numLines];
float offset = 8;  // Set space between lines

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(10);
  for (int i = 0; i < numLines; i++) {
    x[i] = i;  // Set initial position
    speed[i] = 0.1 + (i / offset);  // Set initial speed
  }
}

void draw() {
  background(204);
  for (int i = 0; i < x.length; i++) {
    x[i] += speed[i];  // Update line position
    if (x[i] > (width + offset)) {  // If off the right,
      x[i] = -offset * 2;  // return to the left
    }
    float y = i * offset;  // Set y-coordinate for line
    line(x[i], y, x[i]+offset, y+offset);  // Draw line
  }
}
```

Storing mouse data

Arrays are often used to store data from the mouse. The `pmouseX` and `pmouseY` variables store the cursor coordinates from the previous frame, but there's no built-in way to access the cursor values from earlier frames. At every frame, the `mouseX`, `mouseY`, `pmouseX`, and `pmouseY` variables are replaced with new numbers and their previous numbers are discarded. Creating an array is the easiest way to store the history of these values. In the following example, the most recent 100 values from `mouseY` are stored in a array and displayed on screen as a line from the left to the right edge of the screen. At each frame, the values in the array are shifted to the right and the newest value is added to the beginning.

33-14

```
int[] y;

void setup() {
  size(100, 100);
  y = new int[width];
}

void draw() {
  background(204);
  // Shift the values to the right
  for (int i = y.length-1; i > 0; i--) {
    y[i] = y[i-1];
  }
  // Add new values to the beginning
  y[0] = constrain(mouseY, 0, height-1);
  // Display each pair of values as a line
  for (int i = 1; i < y.length; i++) {
    line(i, y[i], i-1, y[i-1]);
  }
}
```

Apply the same code simultaneously to the `mouseX` and `mouseY` values to store the position of the cursor. Displaying these values each frame creates a trail behind the cursor.

33-15

```
int num = 50;
int[] x = new int[num];
int[] y = new int[num];

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  fill(255, 102);
}

void draw() {
  background(0);
  // Shift the values to the right
  for (int i = num-1; i > 0; i--) {
    x[i] = x[i-1];
    y[i] = y[i-1];
  }
```

33-15 cont.

```
    // Add the new values to the beginning of the array
    x[0] = mouseX;
    y[0] = mouseY;
    // Draw the circles
    for (int i = 0; i < num; i++) {
      ellipse(x[i], y[i], i/2.0, i/2.0);
    }
  }
```

The following example produces the same result as the previous one but uses a more efficient technique. Instead of sorting the array elements in each frame, the program writes the new data to the next available array position. The elements in the array remain in the same position once they are written, but they are read in a different order each frame. Reading begins at the location of the oldest data element and continues to the end of the array. At the end of the array, the % operator (p. 45) is used to wrap back to the beginning. This technique is especially useful with larger arrays, to avoid unnecessary copying of data that can slow down a program.

33-16

```
int num = 50;
int[] x = new int[num];
int[] y = new int[num];
int indexPosition = 0;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  fill(255, 102);
}

void draw()  {
  background(0);
  x[indexPosition] = mouseX;
  y[indexPosition] = mouseY;
  // Cycle between 0 and the number of elements
  indexPosition = (indexPosition + 1) % num;
  for (int i = 0; i < num; i++) {
    // Set the array position to read
    int pos = (indexPosition + i) % num;
    float radius = (num-i) / 2.0;
    ellipse(x[pos], y[pos], radius, radius);
  }
}
```

Array functions

Processing provides a group of functions that assist in managing array data. Only four of these functions are introduced here, but more are explained in the Extended Reference included with the software and available online at *www.processing.org/reference*.

The `append()` function expands an array by one element, adds data to the new position, and returns the new array:

```
String[] trees = { "ash", "oak" };                                    33-17
append(trees, "maple");  // INCORRECT! Does not change the array
print(trees);  // Prints "ash oak"
println();
trees = append(trees, "maple");  // Add "maple" to the end
print(trees);  // Prints "ash oak maple"
println();
// Add "beech" to the end of the trees array, and creates a new
// array to store the change
String[] moretrees = append(trees, "beech");
print(moretrees);  // Prints "ash oak maple beech"
```

The `shorten()` function decreases an array by one element by removing the last element and returns the shortened array:

```
String[] trees = { "lychee", "coconut", "fig"};                       33-18
trees = shorten(trees);  // Remove the last element from the array
print(trees);  // Prints "lychee coconut"
println();
trees = shorten(trees);  // Remove the last element from the array
print(trees);  // Prints "lychee"
```

The `expand()` function increases the size of an array. It can expand to a specific size, or if no size is specified, the array's length will be doubled. If an array needs to have many additional elements, it's faster to use `expand()` to double the size than to use `append()` to continually add one value. The following example saves a new `mouseX` value to an array every frame. When the array becomes full, the size of the array is doubled and new `mouseX` values proceed to fill the enlarged array.

```
int[] x = new int[100];  // Array to store x-coordinates              33-19
int count;  // Store the number of array positions

void setup() {
  size(100, 100);
}
```

33-19 cont.

```
void draw() {
  x[count] = mouseX;        // Assign new x-coordinate to the array
  count++;                  // Increment the counter
  if (count == x.length) {  // If the x array is full,
    x = expand(x);          // double the size of x
    println(x.length);      // Write the new size to the console
  }
}
```

Array values cannot be copied with the assignment operator because they are objects. The most common way to copy elements from one array to another is to use special functions or to copy each element individually within a `for` structure. The `arraycopy()` function is the most efficient way to copy the entire contents of one array to another. The data is copied from the array used as the first parameter to the array used as the second parameter. Both arrays must be the same length for it to work in the configuration shown below.

33-20

```
String[] north = { "OH", "IN", "MI" };
String[] south = { "GA", "FL", "NC" };
arraycopy(north, south);  // Copy from north array to south array
print(south);  // Prints "OH IN MI"
println();
String[] east = { "MA", "NY", "RI" };
String[] west = new String[east.length];  // Create a new array
arraycopy(east, west);  // Copy from east array to west array
print(west);  // Prints "MA NY RI"
```

New functions can be written to perform operations on arrays, but arrays behave differently than data types such as `int` and `char`. When an array is used as a parameter to a function, the address (location in memory) of the array is transferred into the function instead of the actual data. No new array is created, and changes made within the function affect the array used as the parameter.

In the following example, the `data[]` array is used as the parameter to `halve()`. The address of `data[]` is passed to the `d[]` array in the `halve()` function. Because the address of `d[]` and `data[]` is the same, they both affect the same data. When changes are made to `d[]` on line 14, these changes are made to the values in `data[]`. The `draw()` function is not used because the calculation is made only once and nothing is drawn to the diplay window.

```
float[] data = { 19.0, 40.0, 75.0, 76.0, 90.0 };
```
33-21

```
void setup() {
  halve(data);
  println(data[0]);  // Prints "9.5"
  println(data[1]);  // Prints "20.0"
  println(data[2]);  // Prints "37.5"
  println(data[3]);  // Prints "38.0"
  println(data[4]);  // Prints "45.0"
}

void halve(float[] d) {
  for (int i = 0; i < d.length; i++) {  // For each array element,
    d[i] = d[i] / 2.0;                  // divide the value by 2
  }
}
```

Changing array data within a function without modifying the original array requires some additional lines of code. In the following example, the array is passed into the function as a parameter, a new array is made, the values from the original array are copied in the new array, changes are made to the new array, and finally the modified array is returned. Like the previous example, the `draw()` function is not used because nothing is drawn to the display window and the calculation is made only once.

```
float[] data = { 19.0, 40.0, 75.0, 76.0, 90.0 };
float[] halfData;
```
33-22

```
void setup() {
  halfData = halve(data);  // Run the halve() function
  println(data[0] + ", " + halfData[0]);  // Prints "19.0, 9.5"
  println(data[1] + ", " + halfData[1]);  // Prints "40.0, 20.0"
  println(data[2] + ", " + halfData[2]);  // Prints "75.0, 37.5"
  println(data[3] + ", " + halfData[3]);  // Prints "76.0, 38.0"
  println(data[4] + ", " + halfData[4]);  // Prints "90.0, 45.0"
}

float[] halve(float[] d) {
  float[] numbers = new float[d.length];  // Create a new array
  arraycopy(d, numbers);
  for (int i = 0; i < numbers.length; i++) {  // For each element,
    numbers[i] = numbers[i] / 2;  // divide the value by 2
  }
  return numbers;  // Return the new array
}
```

Two-dimensional arrays

Data can also be stored and retrieved from arrays with more than one dimension. Using the example from the beginning of this unit, the data points for the star are put into a 2D array instead of two 1D arrays:

points		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	[0]	50	61	83	69	71	50	29	31	17	39
	[1]	18	37	43	60	82	73	82	60	43	37

A 2D array is essentially a list of 1D arrays. It must be declared, then created, and then the values can be assigned just as in a 1D array. The following syntax converts this array to code:

```
int[][] points = { {50,18}, {61,37}, {83,43}, {69,60}, {71,82},          33-23
                   {50,73}, {29,82}, {31,60}, {17,43}, {39,37} };

println(points[4][0]);  // Prints 71
println(points[4][1]);  // Prints 82
println(points[4][2]);  // ERROR! This element is outside the array
println(points[0][0]);  // Prints 50
println(points[9][1]);  // Prints 37
```

This program shows how it all fits together.

```
int[][] points = { {50,18}, {61,37}, {83,43}, {69,60},          33-24
                   {71,82}, {50,73}, {29,82}, {31,60},
                   {17,43}, {39,37} };

void setup() {
  size(100, 100);
  fill(0);
  smooth();
}

void draw() {
  background(204);
  translate(mouseX - 50, mouseY - 50);
  beginShape();
  for (int i = 0; i < points.length; i++) {
    vertex(points[i][0], points[i][1]);
  }
  endShape();
}
```

It's possible to continue and make 3D and 4D arrays by extrapolating these techniques. However, multidimensional arrays can be confusing, and it's often a better idea to maintain multiple 1D or 2D arrays.

Exercises

1. *Create an array to store the y-coordinates of a sequence of shapes. Draw each shape inside* `draw()` *and use the values from the array to set the y-coordinate of each.*
2. *Write a function to multiply the values from two arrays together and return the result as a new array. Print the results to the console.*
3. *Use a 2D array to store the coordinates for a shape of your own invention. Use a* `for` *structure to draw the shape to the display window.*

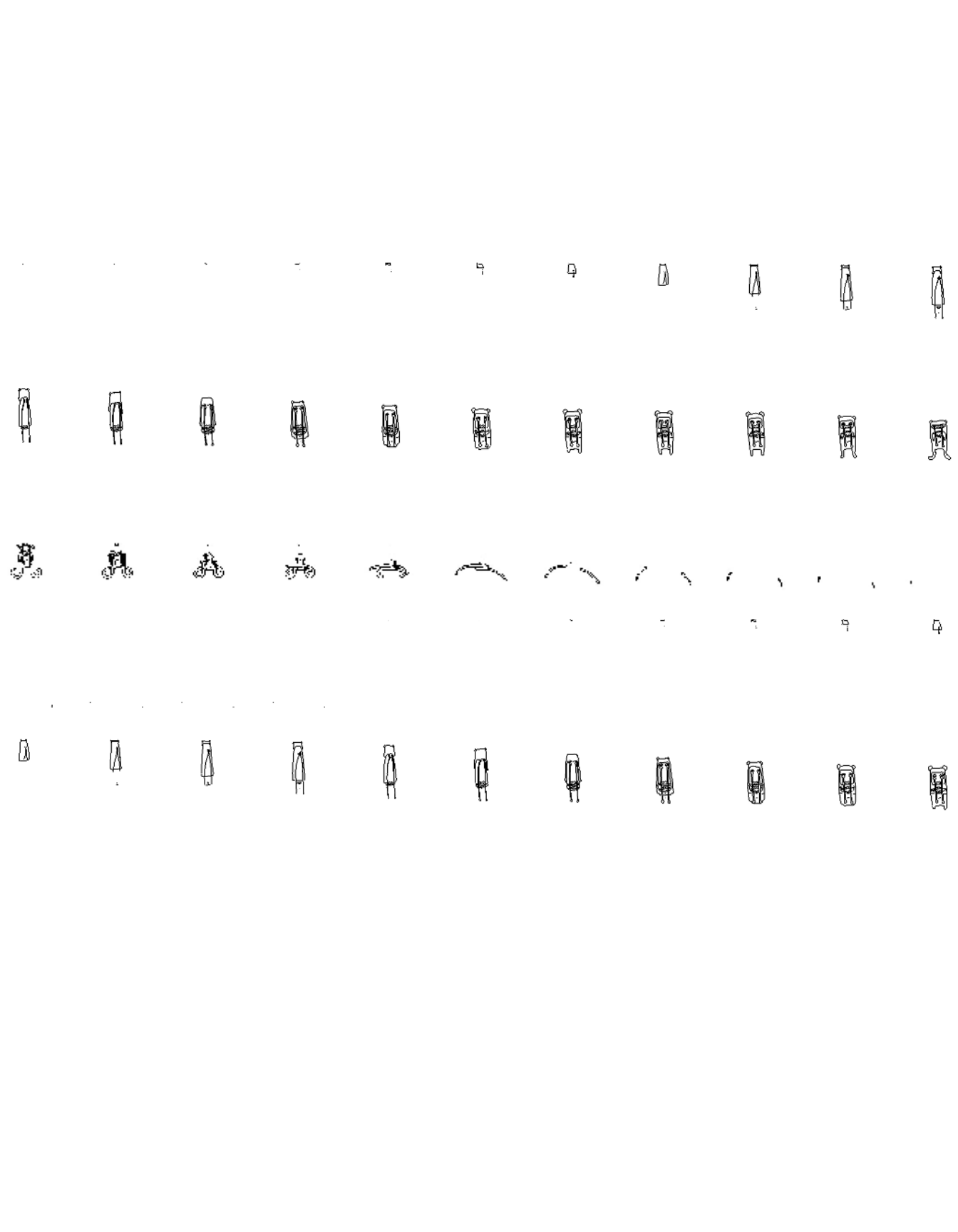


Image 2: Animation

This unit introduces techniques for displaying sequences of images successively, creating animation.

Animation occurs when a series of images, each slightly different, are presented in quick succession. A diverse medium with a century of history, animation has progressed from the initial experiments of Winsor McCay to the commercial and realistic innovations of early Walt Disney studio productions, to the experimental films by such animators as Lotte Reiniger and James Whitney in the mid-twentieth century. The high volume of animated special effects in live-action film and the deluge of animated children's films are changing the role of animation in popular culture.

There's a long history of using software to extend the boundaries of animation. Some of the first computer graphics were presented as animation on film during the 1960s. Because of the cost and expertise required to make these films, they emerged from high-profile research facilities such as Bell Laboratories and IBM's Scientific Center. Kenneth C. Knowlton, then a researcher at Bell Labs, is an important protagonist in the story of early computer animation. He worked separately with artists Stan VanDerBeek and Lillian Schwartz to produce some of the first films made using computer graphics. VanDerBeek and Knowlton's *Poem Field* films, produced throughout the 1960s, utilized Knowlton's BEFLIX code and punch cards to produce permutations of visual micropatterns. Schwartz and Knowlton's *Pixillation* (1970) featured a wide range of effects made by contrasting geometric forms with organic motion. John Whitney worked in collaboration with Jack Citron at IBM to make a number of films including the innovative *Permutations*. This film expresses Whitney's ideas about relationships to music and abstract form by permuting an array of dots into infinite kinetic patterns. Other artists working with computer animation around this time were Larry Cuba, Peter Foldes, and John Stehura.

The paths of contemporary animation and software development often overlap. The 3D visualization of the Death Star in *Star Wars* (1977) was one of the first uses of computer-generated animation in a feature film. Custom software was written to produce a wire-frame fly-through of the massive ship. Interest in computer animation briefly peaked with Disney's *Tron* in 1982, but soon receded due to the film's commercial failure. The industry gradually rebuilt itself into its current role as a major force in contemporary film. Pixar, the hugely successful animation studio that produced *Toy Story* and *The Incredibles*, operated for many years as a software development company. Pixar's RenderMan software (1989) enabled the rendering of 3D computer graphics as photorealistic images. RenderMan became an industry standard and Pixar continues to develop custom software for each film. The success of the company's films reflects its successful marriage of technical virtuosity and masterful storytelling.

Creating unique and experimental animation with software is no longer restricted

to research labs and film studios. The Internet has become a vast repository for experimental software animation. In the late 1990s, *Turux* was created by Lia and Dextro. This online collection of intricate animated images and sounds synthesizes a digital glitch aesthetic with organic qualities. The drawings continually change and sometimes respond to viewer input.

James Paterson (p. 165), a Canadian animator, develops *Presstube.com*, where he produces thousands of drawings, typically organizing tight loops of elements that materialize and dissipate. This technique allows him to arrange these loops in nearly any order while maintaining a fluid progression of growth and decay. The sequences of David Crawford's *Stop Motion Studies* are series of photographs—typically of people in subways in different cities around the world. These photographs are taken in quick succession; presented again in a nonlinear sequence of animated frames, they reveal an incredibly complex and subtle range of human gesture.

Sequential images

Before a series of images can be presented sequentially in a program, all the images must first be loaded. The image variables should be declared outside of `setup()` and `draw()` and then assigned within `setup()`. The following program loads these twelve images from James Paterson . . .

. . . and then draws them to the display window in numeric order.

frame=0

frame=3

frame=11

```
int numFrames = 12;  // The number of animation frames
int frame = 0;  // The frame to display
PImage[] images = new PImage[numFrames];  // Image array

void setup() {
  size(100, 100);
  frameRate(30);  // Maximum 30 frames per second
  images[0]  = loadImage("ani-000.gif");
  images[1]  = loadImage("ani-001.gif");
  images[2]  = loadImage("ani-002.gif");
  images[3]  = loadImage("ani-003.gif");
  images[4]  = loadImage("ani-004.gif");
  images[5]  = loadImage("ani-005.gif");
  images[6]  = loadImage("ani-006.gif");
  images[7]  = loadImage("ani-007.gif");
  images[8]  = loadImage("ani-008.gif");
  images[9]  = loadImage("ani-009.gif");
```

34-01

```
  images[10] = loadImage("ani-010.gif");
  images[11] = loadImage("ani-011.gif");
}

void draw() {
  frame++;
  if (frame == numFrames) {
    frame = 0;
  }
  image(images[frame], 0, 0);
}
```

34-01 cont.

The next example shows an alternative way of loading images by utilizing a `for` structure. These lines of code can load between 1 and 999 images by changing the value of the `numFrames` variable. This shortens the code that flips through each image and returns to the first image at the end of the animation. The `nf()` function (p. 422) on line 11 is used to format the name of the image to be loaded. The names of frames with small numbers are prefaced with zeros so that the images remain in the correct sequence in their folder. For example, instead of *ani-1.gif*, the file is named *ani-001.gif*. The `nf()` function pads the small numbers created in a `for` structure with zeros on the left of the number, so 1 becomes 001, 2 becomes 002, etc. The % operator (p. 45) on line 18 uses the `frameCount` variable to make the `frame` variable increase by 1 each frame and return to 0 once it exceeds 11.

34-02

```
int numFrames = 12;  // The number of animation frames
PImage[] images = new PImage[numFrames];  // Image array

void setup() {
  size(100, 100);
  frameRate(30);  // Maximum 30 frames per second
  // Automate the image loading procedure. Numbers less than 100
  // need an extra zero added to fit the names of the files.
  for (int i = 0; i < images.length; i++) {
    // Construct the name of the image to load
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  // Calculate the frame to display, use % to cycle through frames
  int frame = frameCount % numFrames;
  image(images[frame], 0, 0);
}
```

Displaying the images in random order and for different amounts of time enhances the visual interest of a few frames of animation. Replaying a sequence at irregular intervals, in a random order with random timing, can give the appearance of more different frames than actually exist.

34-03

```
int numFrames = 5;  // The number of animation frames
PImage[] images = new PImage[numFrames];

void setup() {
  size(100, 100);
  for (int i = 0; i < images.length; i++) {
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  int frame = int(random(0, numFrames));  // The frame to display
  image(images[frame], 0, 0);
  frameRate(random(1, 60.0));
}
```

There are many ways to control the speed at which an animation plays. The `frameRate()` function provides the simplest way. Place the `frameRate()` function in `setup()` as seen in the previous example. Use this function to ensure that the software will run at the same speed on other machines.

If you want other elements to move independently of the sequential images, set up a timer and advance the frame only when the timer value grows larger than a predefined value. In the following example, the animation playing in the top of the window is updated each frame and the speed is controlled by the parameter to `frameRate()`. The animation in the bottom frame is updated only twice a second; the timer checks the milliseconds since the last update and changes the frame only if 500 milliseconds (half a second) have elapsed.

34-04

```
int numFrames = 12;   // The number of animation frames
int topFrame = 0;     // The top frame to display
int bottomFrame = 0;  // The bottom frame to display
PImage[] images = new PImage[numFrames];
int lastTime = 0;

void setup() {
  size(100, 100);
  frameRate(30);
  for (int i = 0; i < images.length; i++) {
```

```
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  topFrame = (topFrame + 1) % numFrames;
  image(images[topFrame], 0, 0);
  if ((millis() - lastTime) > 500) {
    bottomFrame = (bottomFrame + 1) % numFrames;
    lastTime = millis();
  }
  image(images[bottomFrame], 0, 50);
}
```

34-04 cont.

Images in motion

Moving one image, rather than presenting a sequence, is another approach to animating images. The same techniques for creating movement presented in Motion 1 (p. 279) apply to images. The following example moves an image from left to right, returning it to the left when it disappears off the edge of the screen.

```
PImage img;
float x;

void setup() {
  size(100, 100);
  img = loadImage("PT-Shifty-0020.gif");
}

void draw() {
  background(204);
  x += 0.5;
  if (x > width) {
    x = -width;
  }
  image(img, x, 0);
}
```

34-05

The transformation functions also apply to images. They can be translated, rotated, and scaled over time to produce motion. In this example, an image is drawn to the center of the display window and rotated slowly around its center.

34-06

```
PImage img;
float angle;

void setup() {
  size(100, 100);
  img = loadImage("PT-Shifty-0023.gif");
}

void draw() {
  background(204);
  angle += 0.01;
  translate(50, 50);
  rotate(angle);
  image(img, -100, -100);
}
```

Images can also be animated by changing their drawing attributes. In this example, the opacity of an image is increased so that it is brought into view over the background.

34-07

```
PImage img;
float opacity = 0;      // Set opacity to the minimum

void setup() {
  size(100, 100);
  img = loadImage("PT-Teddy-0017.gif");
}

void draw() {
  background(0);
  if (opacity < 255) {  // When less than the maximum,
    opacity += 0.5;     // increase opacity
  }
  tint(255, opacity);
  image(img, -25, -75);
}
```

Exercises

1. *Load a sequence of related images into an array and use them to create a linear animation.*
2. *Modify the program for exercise 1 to present each frame of animation at a different rate and in a different sequence.*
3. *Animate an image by changing more than one of its attributes (e.g., size, position, tint).*

Image 3: Pixels

This unit introduces techniques for getting and setting the values for single pixels and groups of pixels.

Syntax introduced:
`get(), set()`

Image 1 (p. 95) defined an image as a rectangular grid of pixels in which each element is a number specifying a color. Because the screen itself is an image, its individual pixels are also defined as numbers. The color values of individual pixels can be read and changed.

Reading pixels

When a Processing program starts, the display window opens at the dimension requested in `size()`. The program gains control over that area of the screen and sets the color value for each pixel. The display window communicates with the operating system, so when the window moves, it takes control of its new area of the screen and gives control of its previous space to the operating system.

The `get()` function can read the color of any pixel in the display window. It can also grab the whole display window or a section of it. There are three versions of this function, one for each use.

```
get()
get(x, y)
get(x, y, width, height)
```

If `get()` is used without parameters, a copy of the entire display window is returned as a `PImage`. The version with two parameters returns the color value of a single pixel at the location specified by the *x* and *y* parameters. A rectangular area of the display window is returned if the additional *width* and *height* parameters are used. If `get()` grabs the entire display window or a section of the window, the returned data must be assigned to a variable of type `PImage`. These images can be redrawn to the screen in different positions and resized.

35-01

```
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage cross = get();  // Get the entire window
image(cross, 0, 50);   // Draw the image in a new position
```

```
smooth();                                                    35-02
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
noStroke();
ellipse(18, 50, 16, 16);
PImage cross = get();           // Get the entire window
image(cross, 42, 30, 40, 40);  // Resize to 40 x 40 pixels
```

```
strokeWeight(8);                                             35-03
line(0, 0, width, height);
line(0, height, width, 0);
PImage slice = get(0, 0, 20, 100);  // Get window section
set(18, 0, slice);
set(50, 0, slice);
```

The `get()` function always grabs the pixels in the display window in the same way, regardless of what is drawn to the window. In the previous examples, the `get()` function grabbed the images of the lines after they had been converted to pixels for display on screen. The following example is the same as code 35-01, but a photograph is first loaded into the display window, so `get()` grabs that image.

```
PImage trees;                                                35-04
trees = loadImage("topanga.jpg");
image(trees, 0, 0);
PImage crop = get();  // Get the entire window
image(crop, 0, 50);   // Draw the image in a new position
```

When used with an x- and y-coordinate, the `get()` function returns values that should be assigned to a variable of the `color` data type. These values can be used to set the color of other pixels or can serve as parameters to `fill()` or `stroke()`. In the following example, the color of one pixel is used to set the color of the rectangle.

```
PImage trees;                                                35-05
trees = loadImage("topanga.jpg");
noStroke();
image(trees, 0, 0);
color c = get(20, 30);  // Get color at (20, 30)
fill(c);
rect(20, 30, 40, 40);
```

The mouse values can be used as the parameters to the `get()` function. This allows the cursor to select colors from the display window. In the following example, the pixel beneath the cursor is read and defines the fill value for the rectangle on the right.

35-06

```
PImage trees;

void setup() {
  size(100, 100);
  noStroke();
  trees = loadImage("topangaCrop.jpg");
}

void draw() {
  image(trees, 0, 0);
  color c = get(mouseX, mouseY);
  fill(c);
  rect(50, 0, 50, 100);
}
```

The `get()` function can be used within a `for` structure to grab many pixels or groups of pixels. In the following example, the values from each row of pixels in the image are used to set the values for the lines on the right. Run this code and move the mouse up and down to see the relation between the image on the left and the bands of color on the right.

35-07

```
PImage trees;
int y = 0;

void setup() {
  size(100, 100);
  trees = loadImage("topangaCrop.jpg");
}

void draw() {
  image(trees, 0, 0);
  y = constrain(mouseY, 0, 99);
  for (int i = 0; i < 49; i++) {
    color c = get(i, y);
    stroke(c);
    line(i+50, 0, i+50, 100);
  }
  stroke(255);
  line(0, y, 49, y);
}
```

Every `PImage` variable has its own `get()` to grab pixels from the image. This allows pixels to be grabbed from an image independently of the pixels in the display window. Because a `PImage` is an object, the `get()` function is accessed with the name of the image and the dot operator. In the following example, the pixels are grabbed directly from the image and not from the screen, so white lines drawn to the display window are not grabbed.

35-08

```
PImage trees;
trees = loadImage("topanga.jpg");
stroke(255);
strokeWeight(12);
image(trees, 0, 0);
line(0, 0, width, height);
line(0, height, width, 0);
PImage treesCrop = trees.get(20, 20, 60, 60);
image(treesCrop, 20, 20);
```

Writing pixels

The pixels in Processing's display window can be written directly with the `set()` function. There are two versions of this function, each with three parameters.

```
set(x, y, color)
set(x, y, image)
```

When the third parameter is a `color`, `set()` changes the color of any pixel in the display window. When the third parameter is an image, `set()` writes an image at the coordinates specified by the *x* and *y* parameters.

35-09

```
color black = color(0);
set(20, 80, black);
set(20, 81, black);
set(20, 82, black);
set(20, 83, black);
```

35-10

```
for (int i = 0; i < 55; i++) {
  for (int j = 0; j < 55; j++) {
    color c = color((i+j) * 1.8);
    set(30+i, 20+j, c);
  }
}
```

The `set()` function can write an image to the display window at any location. Using `set()` to draw an image is faster than using the `image()` function because the pixels are copied directly. However, images drawn with `set()` cannot be resized or tinted, and they are not affected by the transformation functions.

35-11

```
PImage trees;

void setup() {
  size(100, 100);
  trees = loadImage("topangaCrop.jpg");
}

void draw() {
  int x = constrain(mouseX, 0, 50);
  set(x, 0, trees);
}
```

Every `PImage` variable has its own `set()` function to write pixels directly to the image. This allows pixels to be written to an image independently of the pixels in the display window. Because a `PImage` is an object, the `set()` function is run with the name of the image and the dot operator. In the following example, four white pixels are set into the image variable `trees`, and the image is then drawn to the display window.

35-12

```
PImage trees;
trees = loadImage("topangaCrop.jpg");
background(0);
color white = color(255);
trees.set(0, 50, white);
trees.set(1, 50, white);
trees.set(2, 50, white);
trees.set(3, 50, white);
image(trees, 20, 0);
```

Exercises

1. *Load an image and use `get()` to create a collage by overlaying different sections of the same image.*
2. *Load an image and use `mouseX` and `mouseY` to read the value of the pixel beneath the cursor. Use this value to change some aspect of the image.*
3. *Draw a shape in the display window. Copy a section of the window to another by using `get()` and `set()` within a `for` structure.*

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

Typography 2: Motion

This unit introduces typography in motion.

Despite the potential for kinetic type within film, animated typography didn't begin to flourish until the 1950s with the film title work of Saul Bass, Maurice Binder, and Pablo Ferro. These designers and their peers set a high standard with their kinetic title sequences for films such as *North by Northwest* (1959), *Dr. No* (1962), and *Bullitt* (1968). They explored the evocative power of kinetic letterforms to set a mood and express additional layers of meaning in relation to written language. In subsequent years the design of film titles has matured and been augmented by experimental typography for television and the Internet.

Software has played a large role in extending the possibilities of type in motion. The Visual Language Workshop (VLW), founded by Muriel Cooper at the MIT Media Lab in 1985, applied rigorous design thinking to the presentation of kinetic and spatial typography. Researchers including Suguru Ishizaki, Lisa Strausfeld, Yin Yin Wong, and David Small produced progressive typographic explorations ranging from the expression of animated phrases to the navigation of vast typographic landscapes. Because programs did not exist to perform these experiments, the researchers developed custom software to realize their ideas. While at the VLW, David Small created the *Talmud Project* to explore reading in a unique way. It displays the Talmud and related commentaries on the screen simultaneously. A dial controls the legibility of each text through blurring and fading, while keeping each source in context. Peter Cho, building on the explorations of the VLW, wrote software that continued to push the boundaries of expressive kinetic typography. His *Letterscapes* website presents every letter of the Roman alphabet as a character with a unique motion and response in relation to its form. With his *Takeluma* project, Cho went even further by inventing a kinetic alphabet for visualizing speech.

In the last decade, many software tools have been released that facilitate working with kinetic typography. Adobe's Flash software has provided new freedom for working with type on the Web, and Adobe After Effects has supported more sophisticated typography in film and television. This unit introduces techniques for exploring kinetic typography with code.

Words in motion

For typography to move, the program must run continuously, and therefore it requires a `draw()` function. Using typography within `draw()` requires three steps. First, a `PFont` variable must be declared outside of `setup()` and `draw()`. Next, the font should be loaded and set within `setup()`. Finally, the font can be used to place characters on the screen inside `draw()` with the `text()` function.

The following examples use a font named Eureka. To run these examples, you will need to use the "Create Font" tool to create your own font. Change the name of the parameter to `loadFont()` to the name of the font that you created.

36-01

```
PFont font;
String s = "Pea";

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  fill(0);
}

void draw() {
  background(204);
  text(s, 22, 20);
}
```

To put type into motion, simply draw it at a different position each frame. Words can move in an orderly fashion if their position is changed slightly each frame, and they can move without apparent order if placed in an arbitrary position each frame.

36-02

```
PFont font;
float x1 = 0;
float x2 = 100;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  fill(0);
}

void draw() {
  background(204);
  text("Right", x1, 50);
  text("Left", x2, 100);
  x1 += 1.0;
  if (x1 > 100) { x1 = -150; }
  x2 -= 0.8;
  if (x2 < -150) { x2 = 100; }
}
```

36-03

```
PFont font;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  noStroke();
}

void draw() {
  fill(204, 24);
  rect(0, 0, width, height);
  fill(0);
  text("flicker", random(-100, 100), random(-20, 120));
}
```

Typography need not move in order to change over time. More subtle transformations, such as changes in the gray value or transparency of the text, can be made by changing the value of a variable within `draw()`.

36-04

```
PFont font;
int opacity = 0;
int direction = 1;

void setup() {
  size(100, 100);
  font = loadFont("EurekaSmallCaps-36.vlw");
  textFont(font);
}

void draw() {
  background(204);
  opacity += 2 * direction;
  if ((opacity < 0) || (opacity > 255)) {
    direction = -direction;
  }
  fill(0, opacity);
  text("fade", 4, 60);
}
```

Applying the transformations `translate()`, `scale()`, and `rotate()` can also create motion.

36-05

```
PFont font;
String s = "VERTIGO";
float angle = 0.0;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-90.vlw");
  textFont(font, 24);
  fill(0);
}

void draw() {
  background(204);
  angle += 0.02;
  pushMatrix();
  translate(33, 50);
  scale((cos(angle/4.0) + 1.2) * 2.0);
  rotate(angle);
  text(s, 0, 0);
  popMatrix();
}
```

Another technique, called rapid serial visual presentation (RSVP), displays words on the screen sequentially and provides a fundamentally different way to think about reading. Run this program and change the frame rate to see how it affects the process of reading. To store the words within one variable called *words*, this example uses a data element called an Array (explained in Data 4, p. 301).

36-06

```
PFont font;
String[] words = { "Three", "strikes", "and", "you're",
                   "out...", " "  };
int whichWord = 0;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-32.vlw");
  textFont(font);
  textAlign(CENTER);
  frameRate(4);
}
```

```
void draw() {
  background(204);
  whichWord++;
  if (whichWord == words.length) {
    whichWord = 0;
  }
  text(words[whichWord], width/2, 55);
}
```

36-06
cont.

Letters in motion

Individually animated letters offer more flexibility than entire moving words. Building words letter by letter, each with a different movement or speed, can convey a particular meaning or tone. Working in this way requires more patience and often longer programs, but the results can be more rewarding because of the increased possibilities.

36-07

```
// The size of each letter grows and shrinks from
// left to right

PFont font;
String s = "AREA";
float angle = 0.0;

void setup() {
  size(100, 100);
  font = loadFont("EurekaMono-48.vlw");
  textFont(font);
  fill(0);
}

void draw() {
  background(204);
  angle += 0.1;
  for (int i = 0; i < s.length(); i++) {
    float c = sin(angle + i/PI);
    textSize((c + 1.0) * 32 + 10);
    text(s.charAt(i), i*26, 60);
  }
}
```

36-08

```
// Each letter enters from the bottom in sequence and
// stops when it reaches its destination

PFont font;
String word = "rise";
char[] letters;
float[] y;                  // Y-coordinate for each letter
int currentLetter = 0;      // Letter currently in motion

void setup() {
  size(100, 100);
  font = loadFont("EurekaSmallCaps-36.vlw");
  textFont(font);
  letters = word.toCharArray();
  y = new float[letters.length];
  for (int i = 0; i < letters.length; i++) {
    y[i] = 130;             // Position off the screen
  }
  fill(0);
}

void draw() {
  background(204);
  if (y[currentLetter] > 35) {
    y[currentLetter] -= 3; // Move current letter up
  } else {
    if (currentLetter < letters.length-1) {
      currentLetter++;      // Switch to next letter
    }
  }
  // Calculate x to center the word on screen
  float x = (width - textWidth(word)) / 2;
  for (int i = 0; i < letters.length; i++) {
    text(letters[i], x, y[i]);
    x += textWidth(letters[i]);
  }
}
```

Exercises

1. *Select a noun and an adjective. Animate the noun to reveal the adjective.*
2. *Use the transformation functions to animate a short phrase.*
3. *Select a verb and animate each letter of the word to convey its meaning.*

Typography 3: Response

This unit introduces typography that responds to input from the mouse and keyboard.

Many people spend hours a day inputting letters into computers, but this action is very constrained. What features could be added to a text editor to make it more responsive to the typist? For example, the speed of typing could decrease the size of the letters, or a long pause in typing could add many spaces, mimicking a person's pause while speaking. What if the keyboard could register how hard a person is typing (the way a piano plays a soft note when a key is pressed gently) and could automatically assign attributes such as *italics for soft presses* and **bold for forceful presses**? These analogies suggest how conservatively current software treats typography and typing.

Many artists and designers are fascinated with type and have created unique ways of exploring letterforms with the mouse, keyboard, and more exotic input devices. A minimal yet engaging example is John Maeda's *Type, Tap, Write* software, created in 1998 as an homage to manual typewriters. This software uses the keyboard as the input to a black-and-white screen representation of a keyboard. Pressing the number keys cause the software to cycle through different modes, each revealing a playful interpretation of keyboard data. Casey Reas and Golan Levin's *Dakadaka* software from 2000, named after the sounds made while hitting a keyboard, explores the percussive and rhythmic aspects of typing. Input from the keyboard is translated into four different positional abstract alphabets that change according to the speed of typing and the order of the pressed keys. In Jeffrey Shaw and Dirk Groeneveld's *The Legible City* (1989–91), buildings are replaced with three-dimensional letters to create a city of typography that conforms to the streets of a real place. In the Manhattan version, for instance, texts from the mayor, a taxi driver, and Frank Lloyd Wright comprise the city. The image is presented on a projection screen, and the user navigates by pedaling and steering a stationary bicycle situated in front of the projected image. Projects such as these demonstrate that software presents an extraordinary opportunity to extend the way we read and write.

Responsive words

Typographic elements can be assigned behaviors that define a personality in relation to the mouse or keyboard. A word can express aggression by moving quickly toward the mouse, or one moving away slowly can express timidity.

37-01

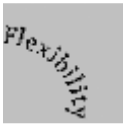
```
// The word "avoid" stays away from the mouse because its
// position is set to the inverse of the cursor position

PFont f;

void setup() {
  size(100, 100);
  f = loadFont("Eureka-24.vlw");
  textFont(f);
  textAlign(CENTER);
  fill(0);
}

void draw() {
  background(204);
  text("avoid", width-mouseX, height-mouseY);
}
```

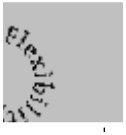
37-02

```
// The word "tickle" jitters when the cursor hovers over

PFont f;
float x = 33;  // X-coordinate of text
float y = 60;  // Y-coordinate of text

void setup() {
  size(100, 100);
  f = loadFont("Eureka-24.vlw");
  textFont(f);
  noStroke();
}

void draw() {
  fill(204, 120);
  rect(0, 0, width, height);
  fill(0);
  // If the cursor is over the text, change the position
  if ((mouseX >= x) && (mouseX <= x+55) &&
      (mouseY >= y-24) && (mouseY <= y)) {
    x += random(-5, 5);
    y += random(-5, 5);
  }
  text("tickle", x, y);
}
```

Responsive letters

Breaking a word into its component letters creates more options in determining its response to the mouse or keyboard. Independent letters that each have the ability to respond in a different way contribute to the word's total response. The following two examples demonstrate this technique. The `toCharArray()` method (p. 108) is used to extract the individual characters from a `String` variable and put them into an array of characters. The `charAt()` method (p. 108) is an alternate way to isolate the individual letters within a String.





37-03

```
// The horizontal position of the mouse determines the
// rotation angle. The angle accumulates with each letter
// drawn to make the typography curve.

String word = "Flexibility";
PFont f;
char[] letters;

void setup() {
  size(100, 100);
  f = loadFont("Eureka-24.vlw");
  textFont(f);
  letters = word.toCharArray();
  fill(0);
}

void draw() {
  background(204);
  pushMatrix();
  translate(0, 33);
  for (int i = 0; i < letters.length; i++) {
    float angle = map(mouseX, 0, width, 0, PI/8);
    rotate(angle);
    text(letters[i], 0, 0);
    // Offset by the width of the current letter
    translate(textWidth(letters[i]), 0);
  }
  popMatrix();
}
```

37-04

```
// Calculates the size of each letter based on the
// position of the cursor so the letters are larger
// when the cursor is closer

String word = "BULGE";
char[] letters;
float totalOffset = 0;
PFont font;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  letters = word.toCharArray();
  textAlign(CENTER);
  fill(0);
}

void draw() {
  background(204);
  translate((width - totalOffset) / 2, 0);
  totalOffset = 0;
  float firstWidth = (width / letters.length) / 4.0;
  translate(firstWidth, 0);
  for (int i = 0; i < letters.length; i++) {
    float distance = abs(totalOffset - mouseX);
    distance = constrain(distance, 24, 60);
    textSize(84 - distance);
    text(letters[i], 0, height - 2);
    float letterWidth = textWidth(letters[i]);
    if (i != letters.length-1) {
      totalOffset = totalOffset + letterWidth;
      translate(letterWidth, 0);
    }
  }
}
```

Exercises

1. *Change the visual attributes of a word as the cursor moves across the display window.*
2. *Draw a verb on screen and have it respond to the cursor to communicate its meaning.*
3. *Select an adverb and a verb. Design the way the verb responds to the mouse to communicate the adverb.*

Color 2: Components

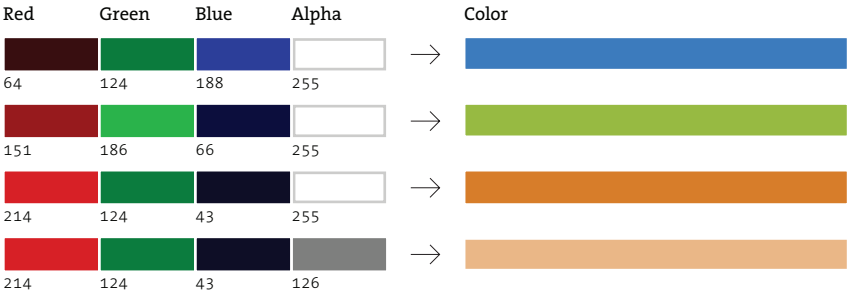
This unit introduces functions for reading the components of a color and discusses techniques for creating dynamic color palettes.

Syntax introduced:
`red(), blue(), green(), alpha(), hue(), saturation(), brightness()`

Colors are stored in software as numbers. Each color is defined by its component elements. When color is defined by RGB values, there are three numbers that store the red, green, and blue components and an optional fourth number that stores a transparency value. When working with HSB values, three numbers store the hue, saturation, and brightness values and a fourth denotes the transparency. The visible color is a combination of these components. Adjusting the individual color properties in isolation from the others is a useful technique for dynamically changing a single color value or the entire palette for a program.

Extracting color

In Processing, the `color` data type is a single number that stores the individual components of a color. This value combines the red, green, blue, and alpha (transparency) components. Behind the scenes, this value is actually an `int`, and can be used interchangeably with an `int` variable anywhere in a program. The `color` data type stores the components of the color as a a series of values from 0 to 255 embedded into this larger number. We can look at an abstracted view of a color with this table:



The `red()`, `green()`, and `blue()` functions are used for reading the components of a color. The `red()` function extracts the red component, the `green()` function extracts the green component, and the `blue()` function extracts the blue component.

```
color c1 = color(0, 126, 255);        // Create a new color
float r = red(c1);                    // Assign 0.0 to r
float g = green(c1);                  // Assign 126.0 to g
float b = blue(c1);                   // Assign 255.0 to b
println(r + ", " + g + ", " + b);     // Prints "0.0, 126.0, 255.0"
color c2 = color(102);                // Create a new gray value
float r2 = red(c2);                   // Assign 102.0 to r2
float g2 = green(c2);                 // Assign 102.0 to g2
float b2 = blue(c2);                  // Assign 102.0 to b2
println(r2 + ", " + g2 + ", " + b2);  // Prints "102.0, 102.0, 102.0"
```

38-01

The `alpha()` function reads the alpha value of the color. Remember, a fourth value added to the `color()` function sets the transparency value for this color. If no alpha value is set, the default 255 is used.

```
color c = color(0, 51, 102);        // Create a new color
color g = color(0, 126, 255, 220);  // Create a new color
float a = alpha(c);                 // Assign 255.0 to a
float b = alpha(g);                 // Assign 220.0 to b
println(a + ", " + b);              // Prints "255.0, 220.0"
```

38-02

The `hue()`, `saturation()`, and `brightness()` functions work like `red()`, `green()`, and `blue()`, but return different components of the color. It makes sense to switch to the HSB color model when using these functions, but sometimes you will want these components while in the default RGB color mode.

```
colorMode(HSB, 360, 100, 100);     // Set color mode to HSB
color c = color(210, 100, 40);     // Create a new color
float h = hue(c);                  // Assign 210.0 to h
float s = saturation(c);           // Assign 100.0 to s
float b = brightness(c);           // Assign 40.0 to b
println(h + ", " + s + ", " + b);  // Prints "210.0, 100.0, 40.0"
```

38-03

```
color c = color(217, 41, 117);     // Create a new color
float r = red(c);                  // Assign 217.0 to r
float h = hue(c);                  // Assign 236.64774 to h
println(r + ", " + h);             // Prints "217.0, 236.64774"
```

38-04

The values from all of these functions are scaled based on the current color mode settings. If the range for color values is changed with `colorMode()`, the values returned will be scaled within the new range.

```
colorMode(RGB, 1.0);              // Sets color mode to RGB
color c = color(0.2, 0.8, 1.0);   // Creates a new color
float r = red(c);                 // Assign 0.2 to r
float h = hue(c);                 // Assign 0.5416667 to h
println(r + ", " + h);            // Prints "0.2, 0.5416667"
```
38-05

The values returned from these color functions are always floating-point values; therefore, you'll receive an error if you try to assign the result to an integer value. If you need the result to be an integer, you can simply convert the value using the `int()` function (p. 107).

```
color c = color(118, 22, 24);  // Create a new color
int r1 = red(c);               // ERROR! red() returns a float
float r2 = red(c);             // Assign 118.0 to r2
int r3 = int(red(c));          // Assign 118 to r3
```
38-06

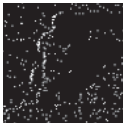
As described in Image 3 (p. 321), these functions make it possible to read the individual color components of the pixels in the display window. In the following examples, the `get()` function is used to access the color at the current cursor position. The components of these colors are extracted and used to set the drawing properties.

```
// Set the stroke color of the lines to the
// red component of the pixel below the cursor

void setup() {
  size(100, 100);
  smooth();
  fill(204, 0, 0);
}

void draw() {
  background(0);
  noStroke();
  ellipse(66, 46, 80, 80);
  color c = get(mouseX, mouseY);
  float r = red(c);  // Extract red component
  stroke(255-r);     // Set the stroke based on red value
  line(mouseX, 0, mouseX, height);
  line(0, mouseY, width, mouseY);
}
```
38-07

38-08

```
// Simulates one pixel of a flat-panel display

PImage wall;

void setup() {
  size(100, 100);
  wall = loadImage("veg.jpg");
  stroke(255);
}

void draw() {
  background(wall);
  color c = get(mouseX, mouseY);
  float r = red(c);      // Extract red
  float g = green(c);    // Extract green
  float b = blue(c);     // Extract blue
  fill(r, 0, 0);
  rect(32, 20, 12, 60);  // Red component
  fill(0, g, 0);
  rect(44, 20, 12, 60);  // Green component
  fill(0, 0, b);
  rect(56, 20, 12, 60);  // Blue component
}
```

Values extracted with the `red()`, `green()`, and `blue()` functions can be used in many different ways. For instance, the numbers can be used to control aspects of motion or the flow of the program. In the following example, the brightness of pixels in an image controls the speed of 400 points moving across the screen. Each point moves across the screen from left to right. The pixel value in the image with the same coordinate as a point is read and used to set the speed at which the point moves. Each point moves slowly through dark areas and quickly through lighter areas. Run the code and try a different photo to see how the same program can be used to create different patterns of motion.



38-09

```
int num = 400;
float[] x = new float[num];
float[] y = new float[num];
PImage img;

void setup() {
  size(100, 100);
  img = loadImage("standing-alt.jpg");
  for (int i = 0; i < num; i++) {
    x[i] = random(width);
    y[i] = random(height);
```

38-09
cont.

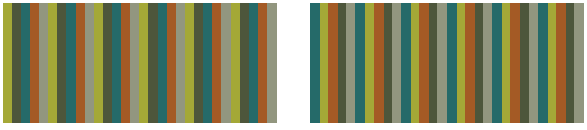
```
  }
  stroke(255);
}

void draw() {
  background(0);
  for (int i = 0; i < num; i++) {
    color c = img.get(int(x[i]), int(y[i]));
    float b = brightness(c) / 255.0;
    float speed = pow(b, 2) + 0.05;
    x[i] += speed;
    if (x[i] > width) {
      x[i] = 0;
      y[i] = random(height);
    }
    point(x[i], y[i]);
  }
}
```

Dynamic color palettes

One of the most important concepts in working with color is relativity. When one color is positioned next to another, they both appear to change. If a color is to appear the same in a new juxtaposition, it often must be physically different (defined with different numbers). This is important to consider when working with color in software, since elements are often moving and changing colors. For example, placing these five colors . . .

A B C D E

. . . in a different order changes their appearance:



ABCDE CADBE

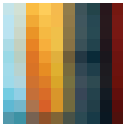
The phenomenon of color relativity can be extended in software by linking colors' relations and making them change dynamically in response to input from the mouse. In the following example, four colors are used. The colors stored in the variables `olive`

and gray remain the same, while the values for yellow and orange change in relation to mouseY and therefore shift as the cursor moves up and down.

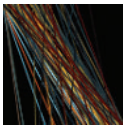
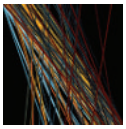
38-10

```
color olive, gray;

void setup() {
  size(100, 100);
  colorMode(HSB, 360, 100, 100, 100);
  noStroke();
  smooth();
  olive = color(75, 61, 59);
  gray = color(30, 17, 42);
}

void draw() {
  float y = mouseY / float(height);
  background(gray);
  fill(olive);
  quad(70 + y*6, 0, 100, 0, 100, 100, 30 - y*6, 100);
  color yellow = color(48 + y*20, 100, 88 - y*20);
  fill(yellow);
  ellipse(50, 45 + y*10, 60, 60);
  color orange = color(29, 100, 83 - y*10);
  fill(orange);
  ellipse(54, 42 + y*16, 24, 24);
}
```

A good technique for creating subtle and complex color palettes with software is to use colors directly from images. Image can be loaded into the software and their colors read using the get() function. For the examples in the rest of this section, the 100 color values from a 10 × 10 pixel image are used to set the fill and stroke colors of shapes. To show the different values more clearly, the image has been enlarged:

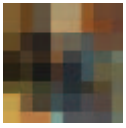


Depending on your goals, you can load a photographic image or one that has been constructed pixel by pixel. An image of any dimension can be loaded and used as a color palette. Sometimes it's appropriate to use only a few colors, and other times hundreds of unique colors might be desired.

38-11

```
PImage img;

void setup() {
  size(100, 100);
  smooth();
  frameRate(0.5);
  img = loadImage("palette10x10.jpg");
}

void draw() {
  background(0);
  for (int x = 0; x < img.width; x++) {
    for (int y = 0; y < img.height; y++) {
      float xpos1 = random(x*10);
      float xpos2 = width - random(y*10);
      color c = img.get(x, y);
      stroke(c);
      line(xpos1, 0, xpos2, height);
    }
  }
}
```



38-12

```
PImage img;

void setup() {
  size(100, 100);
  noStroke();
  img = loadImage("palette10x10.jpg");
}

void draw() {
  int ix = int(random(img.width));
  int iy = int(random(img.height));
  color c = img.get(ix, iy);
  fill(c, 102);
  int xgrid = int(random(-2, 5)) * 25;
  int ygrid = int(random(-2, 5)) * 25;
  rect(xgrid, ygrid, 40, 40);
}
```

Loading the colors from the image into an array opens more possibilities. Once the colors are in an array, they can be easily reordered or shifted. In the following example, the color values from the image are loaded sequentially into an array and then reordered according to their brightness. The `sortColors()` function takes an array of colors as an input, puts them in order from dark to light, and then returns the sorted colors. As it counts from 0 to 255, it puts all the colors with the current value from the unsorted array into the new array.

Original array

Array sorted by brightness

The following example uses the values of the sorted array elements to determine the thickness and center point of the line pairs drawn to the display window. Each pair of lines is spaced evenly at ten-pixel intervals, and a random value is used to access a color from the `imageColors[]` array. Because the colors in the array are sorted, line 24 ensures that the thin lines are bright and the thick lines are dark, regardless of their hue and saturation.

38-13

```
PImage img;
color[] imageColors;

void setup() {
  size(100, 100);
  frameRate(0.5);
  smooth();
  noFill();
  img = loadImage("palette10x10.jpg");
  imageColors = new color[img.width*img.height];
  for (int y = 0; y < img.height; y++) {
    for (int x = 0; x < img.width; x++) {
      imageColors[y*img.height + x] = img.get(x, y);
    }
  }
  imageColors = sortColors(imageColors);
}

void draw() {
  background(255);
  for (int x = 10; x < width; x += 10) {
    int r = int(random(imageColors.length));
    float thick = ((100-r) / 4.0) + 1.0;
```

38-13 cont.

```
    stroke(imageColors[r]);
    strokeWeight(thick);
    line(x, height, x, height-r+thick);
    line(x, 0, x, height-r-thick);
  }
}

color[] sortColors(color[] colors) {
  color[] sorted = new color[colors.length];
  int num = 0;
  for (int i = 0; i <= 255; i++) {
    for (int j = 0; j < colors.length; j++) {
      if (int(brightness(colors[j])) == i) {
        sorted[num] = colors[j];
        num++;
      }
    }
  }
  return sorted;
}
```

Exercises

1. *Write a program to print the red, green, and blue values of every pixel in an image to the console.*
2. *Design a composition that changes based on the `mouseX` value. Make the color for each element of the composition also change in relation to this variable.*
3. *Load an image and use its colors to set the palette for a composition.*

Image 4: Filter, Blend, Copy, Mask

This unit introduces techniques for filtering, blending, copying, and masking images.

Syntax introduced:
`filter(), blend(), blendColor(), copy(), mask()`

Digital images have the remarkable potential to be easily reconfigured and combined with other images. Software now simulates and improves upon complex and time-consuming operations formerly completed in a darkroom with light and chemistry. Every pixel in a digital image is a grouping of numbers that can be added, multiplied, or averaged with the numbers from any other pixel. Some of these calculations are based on simple arithmetic and others use the more complex mathematics of signal processing, but the visual results are most important. Software programs such as the GNU Image Manipulation Program (GIMP) and Adobe's Photoshop have made it possible to perform many of the more common and useful calculations without thinking about the math behind the effects. These programs allow users to easily perform technical operations such as converting images from RGB colors to grayscale values, increasing an image's contrast, or tweaking color balance. Such tools also allow users to apply filters that range from the basic to the kitschy and absurd. A filter might blur an image, mimic solarization, or simulate watercolor effects. The actions of filtering, blending, and copying can easily be controlled with code to produce striking changes. These techniques may be too slow for use in real-time animation.

Filtering, Blending

Processing provides functions to filter and blend images in the display window. Each of these functions operates by transforming the pixel values of a single image or by performing an operation to merge pixels between two different images. The `filter()` function has two prototypes:

```
filter(mode)
filter(mode, level)
```

Eight options exist for the *mode* parameter: THRESHOLD, GRAY, INVERT, POSTERIZE, BLUR, OPAQUE, ERODE, or DILATE. Some of these parameters require the *level* parameter and others don't. For example, the THRESHOLD mode converts every pixel in an image to black or white based on whether its value is above or below the value of the *level* parameter.

The following example applies the THRESHOLD filter to an image with the *level* parameter set to 0.3, which signifies that pixels with a gray value greater than 30 percent

BLUR, 1

BLUR, 4

BLUR, 8

BLUR
Executes a Gaussian blur with the level parameter specifying the extent of the blurring

POSTERIZE, 2

POSTERIZE, 4

POSTERIZE, 8

POSTERIZE
Limits each channel of the image to the number of colors specified as the level parameter

THRESHOLD, 0.2

THRESHOLD, 0.5

THRESHOLD, 0.8

THRESHOLD
Converts the image to black-and-white pixels depending on whether they are above or below the threshold defined by the level parameter

INVERT
Sets each pixel to its inverse value

GRAY
Converts any colors in the image to grayscale equivalents

ERODE
Reduces the light areas with the amount defined by the level parameter

DILATE
Increases the light areas with the amount defined by the level parameter

Filtering

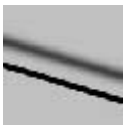
The `filter()` function modifies the pixels of the display window and images. The different kinds of filters seen here provide a range of ready-made options, but it's possible to write custom filters using the language elements introduced in Image 5 (p. 355).

of the maximum brightness will be set to white and pixels below that value will be set to black.

```
PImage img = loadImage("topanga.jpg");
image(img, 0, 0);
filter(THRESHOLD, 0.3);
```

39-01

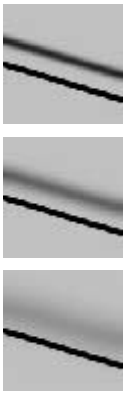
The `filter()` function affects only what has already been drawn. For example, if a program draws two lines and blur is created after one line is drawn, it does not affect the second line:



```
smooth();
strokeWeight(5);
noFill();
line(0, 30, 100, 60);
filter(BLUR, 3);
line(0, 50, 100, 80);
```

39-02

Changing the parameter value of `filter()` with each frame creates movement. The effects of `filter()` are reset each time through `draw()`, but increasing or decreasing the *`level`* parameter results in the filter becoming more or less pronounced as the program runs:



```
float fuzzy = 0.0;

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(5);
  noFill();
}

void draw() {
  background(204);
  if (fuzzy < 16.0) {
    fuzzy += 0.05;
  }
  line(0, 30, 100, 60);
  filter(BLUR, fuzzy);
  line(0, 50, 100, 80);
}
```

39-03

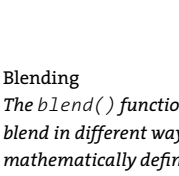
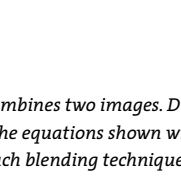
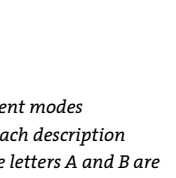
ADD
Additive blending with maximum value of white:
`C = min(A*factor + B, 255)`

SUBTRACT
Subtractive blending with minimum value of black:
`C = max(B - A*factor, 0)`

LIGHTEST
The lightest color is used:
`C = max(A*factor, B)`

DARKEST
The darkest color is used:
`C = min(A*factor, B)`

MULTIPLY
Multiply the colors; result will always be darker:
`C = A * B`

A B C

Blending
The `blend()` function combines two images. Different modes blend in different ways. The equations shown with each description mathematically define each blending technique. The letters A and B are the pixels of the source images, and C is the pixels of the resulting image. The factor is the alpha component (transparency) of the source image. Additional blend modes are documented in the Processing reference.

The PImage class has a filter() method that can isolate filtering to a specific image. The following examples show how to use this method on individual images without affecting the display window.

39-04

```
PImage img = loadImage("forest.jpg");
image(img, 0, 0);
img.filter(INVERT);
image(img, 50, 0);
```

The blend() function mixes pixels in different ways depending on the mode parameter. The blend() function has two different versions.

```
blend(x, y, width, height, dx, dy, dwidth, dheight, mode)
blend(srcImg, x, y, width, height, dx, dy, dwidth, dheight, mode)
```

The *mode* parameter can be BLEND, ADD, SUBTRACT, DARKEST, LIGHTEST, DIFFERENCE, EXCLUSION, MULTIPLY, SCREEN, OVERLAY, HARD_LIGHT, SOFT_LIGHT, DODGE, and BURN. The *x* and *y* parameters are the x- and y-coordinates of the region to copy. The *width* and *height* parameters set the size of the source area. The *dx* and *dy* parameters are the x- and y-coordinate of the destination area. The *dwidth* and *dheight* are the width and height of the destination area. To blend between two images instead of the display window, a second image can be used as the *srcImg* parameter. If the source and destination regions are different sizes, the pixels will be automatically resized to fit the specified target region.

You can blend the image in the display window with itself using any of the different modes. The next example demonstrates blending the window using the ADD mode.

39-05

```
background(0);
stroke(153);
strokeWeight(24);
smooth();
line(44, 0, 24, 80);
line(0, 24, 80, 44);
blend(0, 0, 100, 100, 16, 0, 100, 100, ADD);
```

You can also blend imported images with the display window by including a source image as the first parameter to blend. In this example, the image is seen only through the drawn lines because the background was set to black and the *mode* parameter is DARKEST.

39-06

```
PImage img = loadImage("topanga.jpg");
background(0);
stroke(255);
strokeWeight(24);
```

```
smooth();
line(44, 0, 24, 80);
line(0, 24, 80, 44);
blend(img, 0, 0, 100, 100, 0, 0, 100, 100, DARKEST);
```

39-06 cont.

The `PImage` class has a `blend()` method that can be used to blend an image or two images together without affecting the display window. The following example blends the center of the image *forest.jpg* with the center of *airport.jpg* and then displays the modified image variable to the display window.

```
PImage img = loadImage("forest.jpg");
PImage img2 = loadImage("airport.jpg");
img.blend(img2, 12, 12, 76, 76, 12, 12, 76, 76, ADD);
image(img, 0, 0);
```

39-07

The `blendColor()` function is used to blend individual color values.

```
blendColor(c1, c2, mode)
```

The *c1* and *c2* parameters are the color values that create a new color when blended together. The options for the *mode* parameter are the same as the options for the `blend()` function. Because this unit is printed in black and white, it's not possible to use examples with color values, so the effect is demonstrated in the following example by using gray values.

```
color g1 = color(102);  // Middle gray
color g2 = color(51);   // Dark gray
color g3 = blendColor(g1, g2, MULTIPLY);  // Create black
noStroke();
fill(g1);
rect(50, 0, 50, 100);  // Right rect
fill(g2);
rect(20, 25, 30, 50);  // Left rect
fill(g3);
rect(50, 25, 20, 50);  // Overlay rect
```

39-08

The Processing language includes syntax that makes it easy to write additional custom filters and blending operations. These actions are is discussed further in ƒa (p. 337).

Copying pixels

The `copy()` function has two versions, each of which has a large number of parameters:

```
copy(x, y, width, height, dx, dy, dwidth, dheight)
copy(srcImg, x, y, width, height, dx, dy, dwidth, dheight)
```

The version of `copy()` with eight parameters replicates a region of pixels from the display window in another area of the display window. The version with nine parameters copies all or a portion of the image specified by the *`srcImg`* parameter into the display window. If the source and destination regions are of different sizes, the pixels will automatically be resized to fit the destination width and height. The other parameters are the same as described for `blend()` (p. 351). The `copy()` function differs from the previously discussed `get()` and `set()` functions because it can both get pixels from one location and set them to another. The following two examples demonstrate the function.

39-09

```
PImage img = loadImage("forest.jpg");
image(img, 0, 0);
copy(0, 0, 100, 50, 0, 50, 100, 50);
```

39-10

```
PImage img1, img2;

void setup() {
  size(100, 100);
  img1 = loadImage("forest.jpg");
  img2 = loadImage("airport.jpg");
}

void draw() {
  background(255);
  image(img1, 0, 0);
  int my = constrain(mouseY, 0, 67);
  copy(img2, 0, my, 100, 33, 0, my, 100, 33);
}
```

The `PImage` class also has a `copy()` method. It can be used to copy portions of one image to itself or areas of one image to another. The following example shows this method in action.

```
PImage img = loadImage("tower.jpg");
img.copy(50, 0, 50, 100, 0, 0, 50, 100);
image(img, 0, 0);
```

39-11

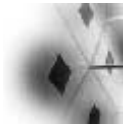
Masking

The `mask()` method of the `PImage` class sets the transparency values of an image based on the contents of another image. The mask image should contain only grayscale data and must be the same size as the image to which it is applied. If the image is not grayscale, it may be converted with the `filter()` function. The light areas of the mask let the original image through, and the dark areas conceal the original. The following example uses `mask()` to composite the images shown below.

airport.jpg

airportmask.jpg

The resulting image and the code to produce it follow:



```
background(255);
PImage img = loadImage("airport.jpg");
PImage maskImg = loadImage("airportmask.jpg");
img.mask(maskImg);
image(img, 0, 0);
```

39-12

Exercises

1. *Load an image and alter it with `filter()`.*
2. *Load three images and combine them with `blend()`.*
3. *Load two images and use `copy()` with `mouseX` and `mouseY` to combine them in a way that reveals the relationship between the images.*

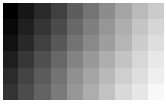
Image 5: Image Processing

This unit introduces techniques for directly accessing the pixels in an image and explains the use of those techniques in modifying images.

Syntax introduced:
`pixels[], loadPixels(), updatePixels(), createImage()`

Image processing is a general term for manipulating and modifying images, whether for the purpose of correcting a defect, improving aesthetic appeal, or facilitating communication. Programs such as GIMP and Adobe Photoshop provide their users with ways to process images including changing the contrast, blurring, and warping. This section explains how some image processing features work to provide a better understanding of their application.

In Processing, each image is stored as a one-dimensional array of colors. When an image is displayed to the screen, each element in the array is drawn as a pixel. The number of elements in the array is determined by multiplying the width of an image by its height. If an image is 100 pixels wide and 100 pixels high, the array will have 10,000 elements. If an image is 200 pixels wide and 2 pixels high, the array will have 400 elements. The first position in the array is the pixel in the upper-left corner of the image, and the last position in the array is the pixel in the lower-right corner. The width and height of an image are used to map each element's position in the one-dimensional array to the two-dimensional position on screen. To make this clear, let's look at an array belonging to a small image of the size 10 × 6 pixels:



When this image is loaded into Processing, its one-dimensional pixel array contains each row of the two-dimensional image, one after another:

Because the image is 10 × 6 pixels, the array has 60 elements. The first element is at position [0] and the last at position [59]. Storing images in this format makes it easy to apply algorithms to the color values.

Pixels

The `pixels[]` array stores a color value for each pixel of the display window. The `loadPixels()` function must be called before the `pixels[]` array is used. After the pixels have been read or changed, they must be updated using the `updatePixels()` function. Like `beginShape()` and `endShape()`, `loadPixels()` and `updatePixels()` should always appear together.

The following example changes the color of the pixels in the display window by changing one pixel each frame based on the current second value. Over time, the pixels are set in order from left to right and top to bottom. The shift from white to black happens with each minute—when the value from `seconds()` jumps from 59 to 0. When the last pixel in the array is set, the program starts again at the beginning of the array.

40-01

```
void setup() {
  size(100, 100);
}

void draw() {
  float gray = map(second(), 0, 59, 0, 255);
  color c = color(gray);
  int index = frameCount % (width*height);
  loadPixels();
  pixels[index] = c;
  updatePixels();
}
```

The `loadPixels()` and `updatePixels()` functions ensure that the `pixels[]` array is ready to be manipulated and that the changes are updated. Be sure to place them around any block of code that manipulates the array, but use them only when necessary because overuse can make your program run slowly.

Reading and writing data directly to and from the `pixels[]` array is a different way to perform the same action as `get()` and `set()`. The x-coordinate and y-coordinate can be mapped to the corresponding position within the array by multiplying the y-coordinate value by the width of the display window and then adding the x-coordinate value. To calculate the location of any pixel in the array, use the equation `(y * width)+x`.

40-02

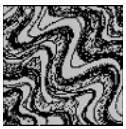
```
// These 3 lines of code are equivalent to: set(25, 50, color(0))
loadPixels();
pixels[50*width + 25] = color(0);
updatePixels();
```

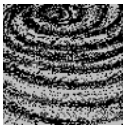
To convert to the opposite direction, divide the pixel's position in the array by the width of the display window to get the y-coordinate, and take the modulo value (p. 45) of the position and the width to get the x-coordinate:

```
// These 3 lines are equivalent to: pixels[5075] = color(0)
int y = 5075 / width;
int x = 5075 % width;
set(x, y, color(0));
```

40-03

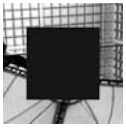
In programs that manipulate many pixels at a time, reading and writing values to the `pixels[]` array is much faster than using `get()` and `set()`. The following two examples show how to achieve the functionality of `get()` and `set()` using the `pixels[]` array. These examples will actually be slower than using `get()` and `set()`, but later examples will be much faster.

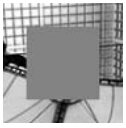




```
void setup() {
  size(100, 100);
}

void draw() {
  // Constrain to not exceed the boundary of the array
  int mx = constrain(mouseX, 0, 99);
  int my = constrain(mouseY, 0, 99);
  loadPixels();
  pixels[my*width + mx] = color(0);
  updatePixels();
}
```

40-04





```
PImage arch;

void setup() {
  size(100, 100);
  noStroke();
  arch = loadImage("arch.jpg");
}

void draw() {
  background(arch);
  // Constrain to not exceed the boundary of the array
  int mx = constrain(mouseX, 0, 99);
  int my = constrain(mouseY, 0, 99);
  loadPixels();
  color c = pixels[my*width + mx];
  fill(c);
  rect(20, 20, 60, 60);
}
```

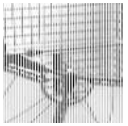
40-05

Each image has its own `pixels[]` array that is accessed with the dot operator. This array makes it possible to change an image while leaving the pixels in other images and the display window untouched. In the next example, pixels inside an image are colored black according to the position of the mouse.

40-06

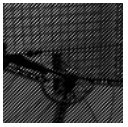
```
PImage arch;

void setup() {
  size(100, 100);
  arch = loadImage("arch.jpg");
}

void draw() {
  background(204);
  int mx = constrain(mouseX, 0, 99);
  int my = constrain(mouseY, 0, 99);
  arch.loadPixels();
  arch.pixels[my*width + mx] = color(0);
  arch.updatePixels();
  image(arch, 50, 0);
}
```

Using the `pixels[]` array rather than the `image()` function to draw the image to the display window provides more control and leaves room for variation in displaying the image. Small calculations modifying the `for` structure and the `pixels[]` array reveal some of the potential of this technique.



40-07

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i += 2) {
  pixels[i] = arch.pixels[i];
}
updatePixels();
```



40-08

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i += 3) {
  pixels[i] = arch.pixels[i];
}
updatePixels();
```

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i++) {
  pixels[i] = arch.pixels[count - i - 1];
}
updatePixels();
```

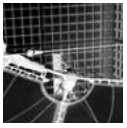
40-09

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i++) {
  pixels[i] = arch.pixels[i/2];
}
updatePixels();
```

40-10

Pixel components

The `red()`, `green()`, and `blue()` functions (pp. 337–338) are used to read the individual color components from each pixel in an image. These components can be changed and then returned to the `pixels[]` array to modify the image. For example, if each value is multiplied by 2, the image will become lighter; if each value is divided by 2, the image will become darker. Using a `for` structure makes it easy to read and change every pixel in the display window. Because the `pixels[]` array is a one-dimensional array, only one `for` structure is necessary to modify every pixel in the image. The following example shows how to invert an image.

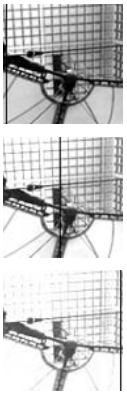


```
PImage arch = loadImage("arch.jpg");
background(arch);
loadPixels();
for (int i = 0; i < width*height; i++) {
  color p = pixels[i];         // Grab pixel
  float r = 255 - red(p);      // Modify red value
  float g = 255 - green(p);    // Modify green value
  float b = 255 - blue(p);     // Modify blue value
  pixels[i] = color(r, g, b);  // Assign modified value
}
updatePixels();
```

40-11

Values from the keyboard and the mouse can be used to change the way the `pixels[]` array is altered while the program runs. In the following example, a color image is

converted to gray values by averaging its components. These values are incremented by `mouseX` to make the image lighter as the mouse moves to the right.



40-12

```
PImage arch;

void setup() {
  size(100, 100);
  arch = loadImage("arch.jpg");
}

void draw() {
  background(arch);
  loadPixels();
  for (int i = 0; i < width*height; i++) {
    color p = pixels[i];    // Read color from screen
    float r = red(p);       // Modify red value
    float g = green(p);     // Modify green value
    float b = blue(p);      // Modify blue value
    float bw = (r + g + b) / 3.0;
    bw = constrain(bw + mouseX, 0, 255);
    pixels[i] = color(bw);  // Assign modified value
  }
  updatePixels();
  line(mouseX, 0, mouseX, height);
}
```

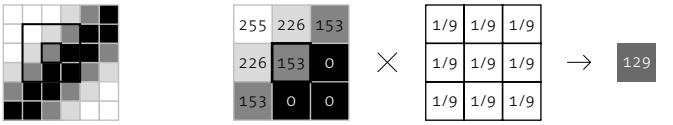
The functions for extracting individual color components are accurate and easy to use, but they are slow. When an idea requires using these functions hundreds or thousands of times each frame, they can be replaced with a technique called bit-shifting (p. 673).

Convolution

Another way to modify an image is to change the value of each pixel in relation to the neighboring pixels. This process operates in a similar way as the cellular automata introduced in Simulate 1 (p. 461). A matrix of numbers called a convolution kernel is applied to every pixel in the image—neighboring pixels are multiplied by the corresponding kernel value and added together to set the value of the center pixel. Applying the kernel to every pixel in the image is called convolving the image. This type of math can be performed very efficiently, and in advanced programs such as Photoshop, most of the filters are implemented in this manner.

As an example, let's use a kernel to determine the value for the pixel at coordinate (2,2) in the simple 6 × 6 pixel image shown below. The center of the kernel is first placed at the coordinate, and then each value within the area is multiplied by the

corresponding kernel value and all are added together. The sum is the new value for the pixel at the center of the kernel:



Original image | Enlarged region and gray values | Kernel values | Result

The first expression created by multiplying the image gray values by the kernel values is

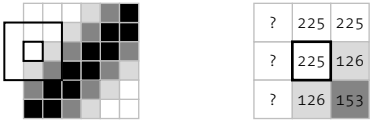
```
(255 * 0.111) + (226 * 0.111) + (153 * 0.111) +
(226 * 0.111) + (153 * 0.111) + (  0 * 0.111) +
(153 * 0.111) + (  0 * 0.111) + (  0 * 0.111)
```

This simplifies to

```
28.305 + 25.086 + 16.983 +
25.086 + 16.983 +  0.000 +
16.983 +  0.000 +  0.000
```

This simplifies further to 129.426, is converted to the integer value 129, and becomes the gray value of the pixel.

To convolve the entire image, perform this action for all of the pixels in the image. It's clear that a problem arises when you try to use the kernel at the edges of the image. At the edges, there are no adjacent pixels from which to take values:



Original image | Enlarged region and gray values

To simplify the code in the examples below, this is ignored and only the pixels away from the border are used.

Patterns in the kernel numbers create different types of filters. If all the values for the kernel are positive, it creates what is called a low pass filter. A low pass filter removes areas where extreme differences in the values of adjacent pixels exist. For example, if one pixel is white and the next is black, they will create a gray when averaged together. When applied to an entire image, a low pass filter causes a blur. A mixture of positive and negative values can be used to create a high pass filter. A high pass filter removes areas where there is little difference in value between adjacent pixels. This technique sharpens images. A specific type of high pass filter is used for edge detection. An edge is an area that contains sudden changes in value. Common kernels for edge detection have negative numbers along one side and positive numbers on the opposing side, with zeros in the middle. For all kernels, the sum of the values must be 1 for the brightness to

remain the same when the image is convolved. If the sum is a smaller or larger number, the image will become darker or lighter in value than the original.

The following example demonstrates how to use a 3 × 3 kernel matrix to transform an image. Modify the values in the `kernel[][]` array to try different filter techniques. There are a few samples on the adjacent page. The `createImage()` function creates an empty pixel buffer. The function requires three parameters that assign the width, height, and format of the image. The format can be RGB (full color) or ARGB (full color with alpha). It is not necessary to use `loadPixels()` immediately after `createImage()`.

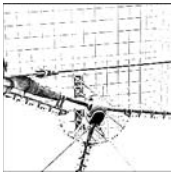
40-13

```
float[][] kernel = { { -1, 0, 1 },
                     { -2, 0, 2 },
                     { -1, 0, 1 } };
size(100, 100);

PImage img = loadImage("arch.jpg");  // Load the original image
img.loadPixels();
// Create an opaque image of the same size as the original
PImage edgeImg = createImage(img.width, img.height, RGB);

// Loop through every pixel in the image.
for (int y = 1; y < img.height-1; y++) {  // Skip top and bottom edges
  for (int x = 1; x < img.width-1; x++) { // Skip left and right edges
    float sum = 0;  // Kernel sum for this pixel
    for (int ky = -1; ky <= 1; ky++) {
      for (int kx = -1; kx <= 1; kx++) {
        // Calculate the adjacent pixel for this kernel point
        int pos = (y + ky)*width + (x + kx);
        // Image is grayscale, red/green/blue are identical
        float val = red(img.pixels[pos]);
        // Multiply adjacent pixels based on the kernel values
        sum += kernel[ky+1][kx+1] * val;
      }
    }
    // For this pixel in the new image, set the gray value
    // based on the sum from the kernel
    edgeImg.pixels[y*img.width + x] = color(sum);
  }
}
// State that there are changes to edgeImg.pixels[]
edgeImg.updatePixels();
image(edgeImg, 0, 0);  // Draw the new image
```

```
.11  .11  .11
.11  .11  .11
.11  .11  .11
```

```
.11  .11  .11
.11  .66  .11
.11  .11  .11
```

```
-1   -1   -1
-1    8   -1
-1   -1   -1
```

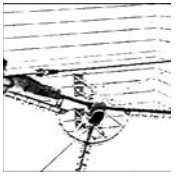


```
-1   -1   -1
-1   12   -1
-1   -1   -1
```

```
-1    0    1
-2    0    2
-1    0    1
```

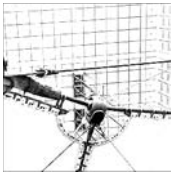
```
-2    0    1
-3    0    2
-2    0    1
```

```
-1   -2   -1
 0    0    0
 1    2    1
```



```
-1   -2   -1
 0    0    0
 2    3    2
```

```
 0   -1    0
-1    4   -1
 0   -1    0
```



```
 0   -1    0
-1    6   -1
 0   -1    0
```

Convolving an image

Eight common 3 × 3 convolution kernels and their effects. A kernel is normalized if the sum of the values is 1. If the sum is above 1, the image becomes lighter, and if it's below 1, the image becomes darker. These numbers can be inserted into code 40-13.

Image as data

This unit has introduced digital images as one-dimensional sequences of numbers that define colors. This numerical data, however, need not be viewed as colors—it can be used to generate motion or define the vertices of a shape. The following examples use the data from the `pixels[]` array of an image to generate alternative representations.

40-14

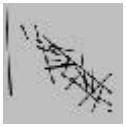
```
// Convert pixel values into a circle's diameter

PImage arch;
int index;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
  color c = arch.pixels[index];  // Get a pixel
  float r = red(c) / 3.0;  // Get the red value
  ellipse(width/2, height/2, r, r);
  index++;
  if (index == width*height) {
    index = 0;  // Return to the first pixel
  }
}
```

40-15

```
// Convert the red values of pixels to line lengths

PImage arch;

void setup() {
  size(100, 100);
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
```

```
  int my = constrain(mouseY, 0, 99);
  for (int i = 0; i < arch.height; i++) {
    color c = arch.pixels[my*width + i];  // Get a pixel
    float r = red(c);  // Get the red value
    line(i, 0, i, height/2 + r/6);
  }
}
```

40-15 cont.

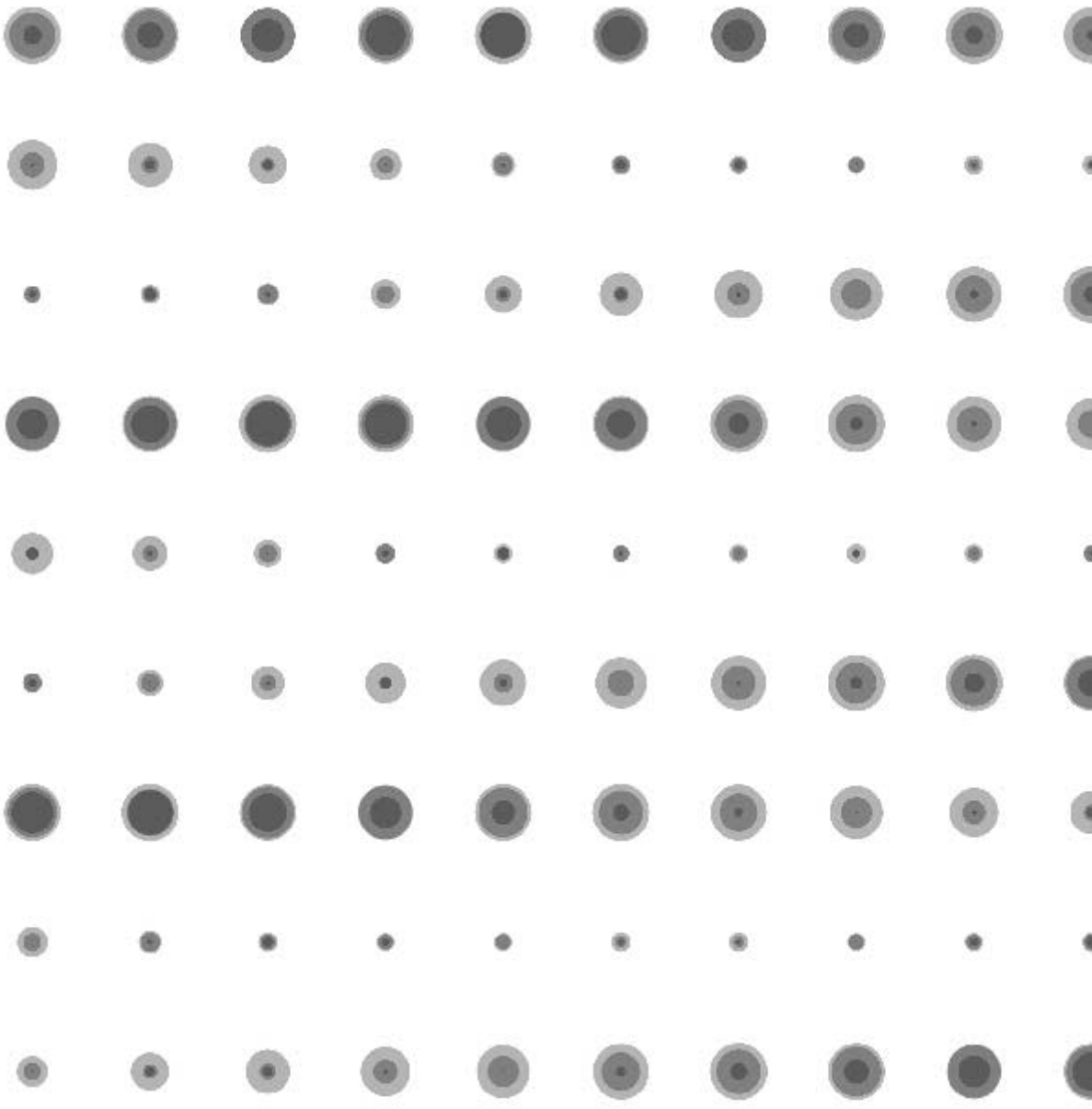


```
// Convert the blue values from one row of the image
// to the coordinates for a series of lines

PImage arch;

void setup() {
  size(100, 100);
  smooth();
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
  int mx = constrain(mouseX, 0, arch.width-1);
  int offset = mx * arch.width;
  beginShape(LINES);
  for (int i = 0; i < arch.width; i += 2) {
    float r1 = blue(arch.pixels[offset + i]);
    float r2 = blue(arch.pixels[offset + i + 1]);
    float vx = map(r1, 0, 255, 0, height);
    float vy = map(r2, 0, 255, 0, height);
    vertex(vx, vy);
  }
  endShape();
}
```

40-16

Exercises

1. *Write your own image filter by modifying the values of* `pixels[]`.
2. *Explore different kernels to convolve an image and write a program to display your most interesting discovery.*
3. *Load an image and use its data to generate an animation that reflects the original image.*



Output 1: Images

This unit explains how to save images and sequences of images while a program is running.

Syntax introduced:
`save(), saveFrame()`

A computer screen displays a new image to the screen many times each second. Most operating systems provide a way to capture these images while the computer is running. If you're using a Macintosh computer, press Command-Shift-3. In Windows, press the Print Screen (Prnt Scrn) key on the keyboard to save the image to the clipboard and then open an image editor and paste it into a window. There are also software applications that facilitate saving the images from the screen.

Saving images from a software application can be useful as a documentation technique or as a way to create frame-by-frame animation. The rate at which software can draw to the screen is always limited by the speed of the computer. When software is intended to be viewed live, the image quality often has to be reduced because of the need to draw many frames each second. But if the software is for still images and animation, each image can draw over a period of hours or days instead of in 1/30 of a second. This control over time enables the composition of images containing more visual elements or the use of rendering techniques such as blurring. After images are saved, they can be loaded into image editing programs or programs such as Apple's QuickTime Pro software and made into movies.

If you're creating animation, you'll save your files at different sizes depending on where you live. The NTSC video format used widely in the Americas and the PAL format used widely in Europe, Africa, and Asia each have different resolutions and frame rates. If you're making images for television, DVD, or high-definition video, you'll save images at different sizes and rates. The following table shows the basic resolutions and frames per second for different formats:

Format	FPS	DVD Resolution	HDTV
NTSC	30	720 × 480	1280 × 720 or 1920 × 1080
PAL	25	720 × 576	1280 × 720 or 1920 × 1080

For example, making a 30-second animation for an NTSC DVD requires 30 frames each second for a total of 900 frames. Producing the same animation for the PAL format requires 25 frames each second for a total of 750 frames. Displaying an animation on the Web requires deciding how large the file should be. The pixel dimensions of the file and its format affect the size of the file download. Animation files stored online should be as small as possible to make them fast to download. Make your decisions regarding the pixel dimensions and compression formats by considering the type of content, how

much space you have to store the files, and how comfortable you are with degrading the image quality by compressing the image. Regardless of the format or delivery of your animations, the process involves saving a series of frames, loading them into a separate application, and saving them as a movie. Saving images at a higher resolution is discussed in the Extension 6 (p. 603).

Saving images

The `save()` function saves an image of the display window. It requires one parameter, a `String` that becomes the name of the saved image file:

```
save(filename)
```

Images are saved in a variety of formats depending on the extension used in the filename parameter. For example, the *`filename`* parameter *myFile.tif* will save a TIFF file, and the value *myFile.tga* will save a TARGA file. If no extension is included in the filename, the image will save as TIFF and *.tif* will be added to the name. Be sure to remember to put the name of the file in quotes to distinguish it as a `String`. The image is saved into the current sketch's folder.

```
line(0, 0, width, height);
line(width, 0, 0, height);                                    41-01
// Saves the TIFF file "x.tif" to the current sketch's folder
save("x.tif");
```

Only the elements drawn before `save()` will be included in the image; those drawn afterward will not. In this example, only the first line is saved in the file *line.tif* but both lines are displayed on the screen.

```
line(0, 0, width, height);
// Saves the TIFF file "line.tif" to the current sketch's folder   41-02
save("line.tif");
line(width, 0, 0, height);
```

If the `save()` function appears within `draw()`, the file is continually rewritten each time `draw()` is run. The file saved during the previous frame is replaced with a file from the current frame. This is avoided by putting `save()` within an event such as `mousePressed()` or `keyPressed()`. Because these events are always called when `draw()` is finished, the saved image will include everything from the frame that was drawing when the event occurred. In the following example, the file *line.tif* is saved when a mouse button is pressed. If a mouse button is pressed more than once, the file is created again and the original file is removed.

41-03

```
void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  line(0, 0, mouseX, height);
  line(width, 0, 0, mouseY);
}

void mousePressed() {
  save("line.tif");
}
```

Saving sequential images

The `saveFrame()` function saves a numbered sequence of images:

```
saveFrame()
saveFrame("filename-####.ext")
```

If `saveFrame()` is used without a parameter, it saves the files as *screen-0000.tif*, *screen-0001.tif*, *screen-0002.tif*, etc. The *`filename-`* component can be changed to any name, and the *`.ext`* component can be set to *.tif* or *.tga* to set the format. The *`####`* portion of the name specifies the number of digits. When the files are saved, the four #'s are replaced with the value of the `frameCount` variable (p. 173). For example, the 127th frame will be called *filename-0127.tif* and the 1732nd frame will be called *filename-1732.tif*. Add an extra # symbol to support 10,000 frames or more.

41-04

```
// Save the first 50 frames

float x = 33;
float numFrames = 50;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(0);
  x += random(-2, 2);
```

```
  ellipse(x, 50, 40, 40);                                          41-04
  if (frameCount <= numFrames) {                                   cont.
    saveFrame("circles-####.tif");
  }
}
```

Using `saveFrame()` inside an `if` structure allows the program to save images only if a certain condition is met. For example, you may want to save a sequence of 200 frames after the mouse is pressed. Or you may want to save one frame and then skip a few before saving another. The following code fragments present ways to achieve similar objectives.

```
// Save 24 frames, from x-1000.tif to x-1023.tif                    41-05
void draw() {
  background(204);
  line(mouseX, mouseY, pmouseX, pmouseY);
  if ((frameCount > 999) && (frameCount < 1024)) {
    saveFrame("x-####.tif");
  }
}
```

```
// Save every fifth frame (i.e., x-0005.tif, x-0005.tif, x-0010.tif)  41-06
void draw() {
  background(204);
  line(mouseX, mouseY, pmouseX, pmouseY);
  if ((frameCount % 5) == 0) {
    saveFrame("x-####.tif");
  }
}
```

Exercises

1. *Save an image from one of your previously created programs.*
2. *Save a sequence of images from one of your previously created programs.*
3. *Use another application to create a movie from the frames saved in exercise 2.*

Synthesis 3: Motion and Arrays

This unit presents examples that synthesize concepts from Motion 1 through Output 1.

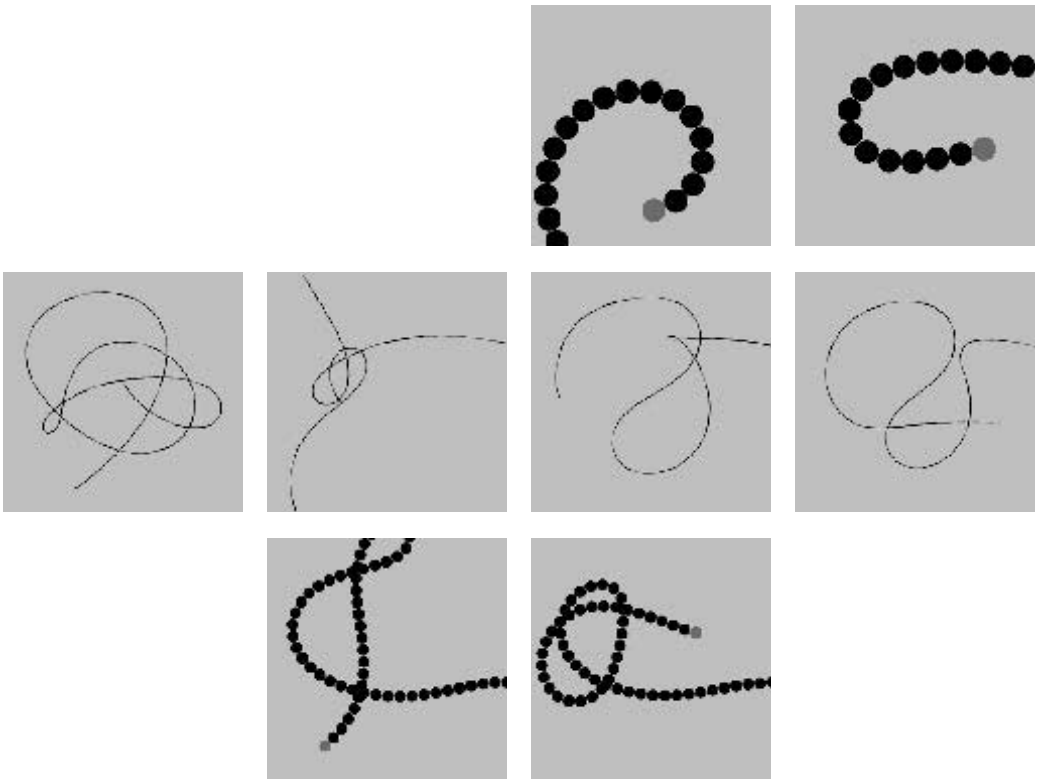
The previous units introduced concepts and techniques including motion, image processing, color components, exporting images, and arrays. Each of these topics opens a broad area for exploration, and they can be combined to create even more options. This unit elaborates on motion and arrays.

It's easy to create simple motion with software, but sophisticated movement requires thought and skill. The units Motion 1 (p. 279) and Motion 2 (p. 291) introduce the topics of nonlinear motion, moving on curves, moving with sinusoids, and integrating unpredictability. More complicated and believable movement requires combining these ideas. Because computers are machines and code is highly structured, it's easier to create mechanical motion than organic motion. Creating believable organic motion is one of the most difficult challenges in programming movement.

Arrays are one of the more difficult software concepts to digest, but they are essential for managing programs with many elements. For example, arrays can make writing some of the programs in Synthesis 1 and Synthesis 2 easier. The collage software (p. 150), for example, loads 29 images into separate variables and has 8 separate lines of code to display each. Making an array of `PImage` variables would improve the code's modularity so it would be easier to maintain and change. The typing program (p. 258) could also be greatly enhanced with arrays. For each line of text, a separate string and group of variables is necessary for saving the angle, position, and size of each line. Using an array for each variable would make it possible to include more lines of text without increasing the length of the program.

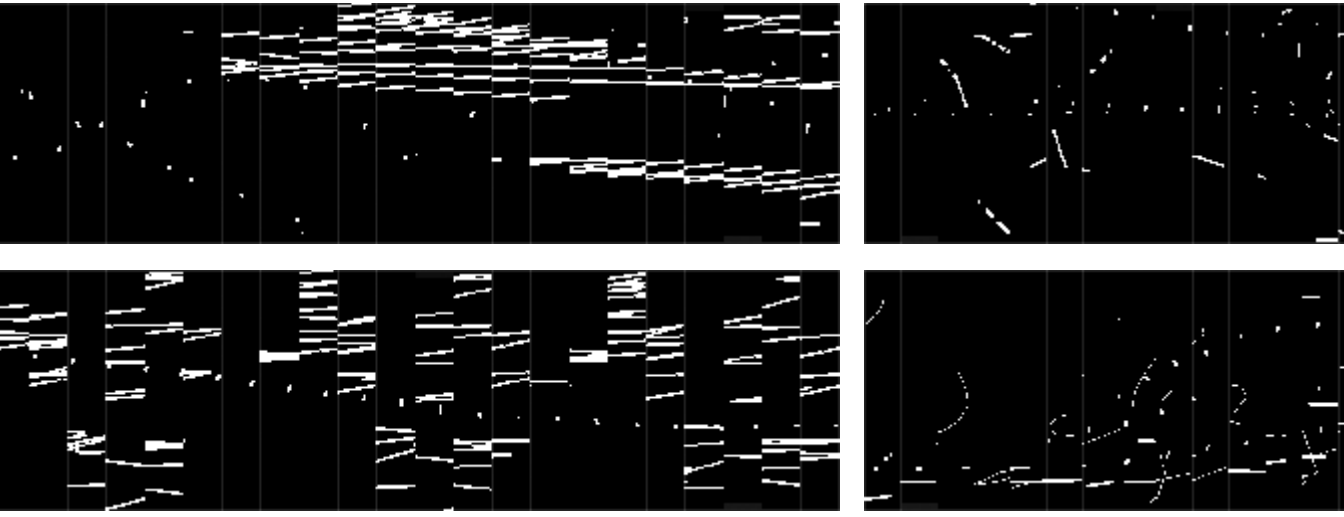
Three of the four examples in this unit use arrays to demonstrate their use. Two of the programs focus on utilizing arrays to create motion, one combines drawing with an array of images, and the fourth adds images together to create a software puppet that responds to the mouse.

The four programs presented here were written by different programmers. Unlike most of the other examples in the book, which have been written in a similar style, each of these programs reflects the personal programming style of its author. Learning how to read programs written by other people is an important skill.
The software featured in this unit is longer than the brief examples that fill this book. It's not practical to print it on these pages, but the code is included in the Processing code download at www.processing.org/learning.



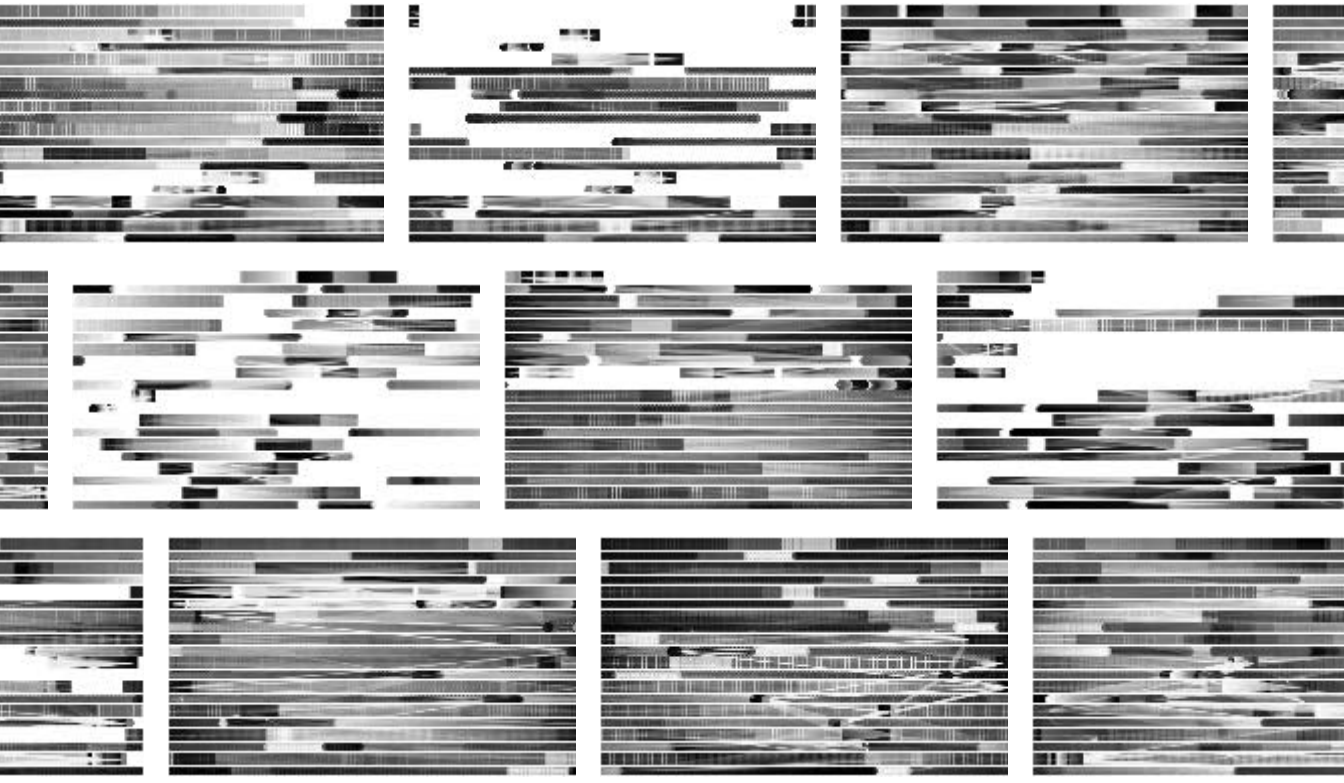
Centipede. The head of this chain of circles is controlled by the cursor. The position of each circle is stored in two arrays. One array stores the x-coordinates and the other stores the y-coordinates. The position of the head is updated every frame based on the current cursor location, and the position of each following circle is calculated based on its position in relation to the circle preceding it. Within the code, change the values of the `n_nodes` variable to set the number of elements in the chain, and change the value of the `node_length` variable to set the size of each element.

Program written by Ariel Malka (www.chronotext.org)



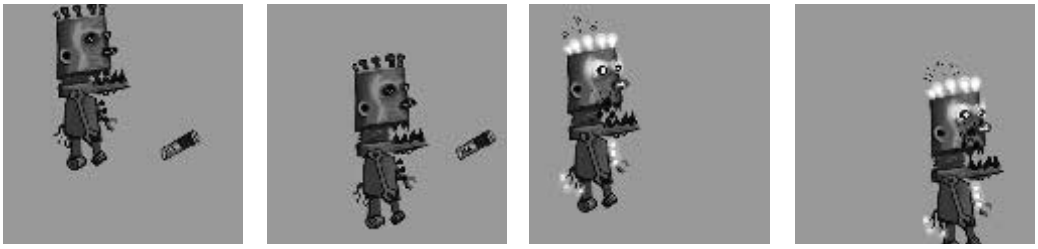
Chronodraw. Inspired by the photographic explorations of Eadweard Muybridge (p. 295), this software simultaneously shows lines drawn at different times on a single frame. An array stores 200 images, with 23 displayed on screen at one time. Lines are drawn directly into an image array using the custom drawing function. The images in the array are displayed to the screen in the order defined by the variables, which are set by the markers on the top and bottom of the display window. The bottom marker sets the speed and the top marker sets the number of images skipped between adjacent units. Press the spacebar and move the mouse left and right to change the markers.

Program written by Andreas Gysin (www.ertdfgcvb.ch)



AmoebaAbstract_03. The structured and layered textures in these images are created by moving rows of circles and squares across the display window. Six arrays are used to store the information for each element. They store the x-coordinate, y-coordinate, and speed and the red, green, and blue color data. Each element is mostly transparent, and each frame accumulates with the previous frames to create a dynamic blend of the different hues. Click the mouse to create a new color palette. Change the position of the mouse to alter the speed and direction of the elements.

Program written by Marius Watz (www.unlekker.net)



Mr. Roboto. This robot is comprised of a series of images. The pieces can move independently to create an articulated character with fluid movements. Using motion from the `sin()` function, the robot's mouth opens and closes as it continuously moves toward the battery. The position of the cursor controls the position of the battery. When the battery comes in contact with the robot, the robot's mouth crushes the battery. This effect is created by swapping the battery and robot images with new images.

Program written by Leon Hong (www.jigobite.com)

Still image from the R.E.M. “Animal” music video, 2003. Image courtesy of **Motion Theory, Inc.**

Interviews 3: Animation, Video

Motion Theory. R.E.M. “Animal”
Bob Sabiston. *Waking Life*
Jennifer Steinkamp. *Eye Catching*
Semiconductor. *The Mini-Epoch Series*

R.E.M. "Animal" *(Interview with Mathew Cullen and Grady Hall)*

Creators	Motion Theory (Mathew Cullen and Gray Hall, directors)
Year	2003
Medium	Music Video
Software	Processing, Maya, Adobe After Effects
URL	www.motiontheory.com

What is R.E.M. "Animal"?

"Animal" is a music video for the band R.E.M. Mathew Cullen & Grady Hall of Motion Theory directed this video featuring lead singer Michael Stipe. It was shot in Los Angeles and Vancouver, B.C. The video is about an eclipse enabling a man (Stipe) to see the invisible forces of life—emotion, connection, humanity—in the form of visible energy. While the eclipse holds, everything takes on a different quality—energies arise that resemble living constellations, people appear to be connected by bioluminescent strands, and movement creates patterns of light. All of these forces connect with the video's cosmic moments, which take us past planets, galaxies, and nebulae, connecting outer space with inner space. When the eclipse ends, even though we can't see the energy and connections anymore, there's a sense that they're always there, right underneath the surface.

Why did you create R.E.M. "Animal"?

Motion Theory, a micro-studio, was approached by Warner Bros. Records to come up with a treatment for "Animal." Simply put, we created the video because we loved the lyrics and themes of the song. We knew there was a chance to express something in a new way—and that R.E.M. and Michael Stipe would be supportive in creating something different and resonant. Stripping this down to its most basic level, our deepest motivation was the desire to communicate the connections between people in beautiful and poetic ways.

What software tools were used?

We filmed the bulk of the video on location, with some elements of the video shot on greenscreen. We also did an additional effects shoot for some of the practical effects and time-lapse moments. Once we had a finished edit, we had an ambitious task ahead: how to depict the unseen connections between people in a way that felt new and natural, and yet avoided looking like Disney fairy-dust. We used Processing to create the "living constellation" look—we loved the way that it interacted directly with the movement in the film. The software helped incredibly, because hand-animating those moments to the level of fluidity that was achieved would have been very difficult, if not impossible. The planetary sequences, cityscapes, insects, and "bioluminescent arms" were created during long hours with Maya, and everything was composited in After Effects. Most shots in the finished piece combine a bit of everything, but our aim was to make sure that the technology was invisible and simply let the story come through.

Why did you use these tools?

We wanted to create a thoroughly unique look for the energy depicted in the video. Conventional visual effects were simply not sophisticated, delicate, and interactive enough to capture the feeling, subtlety, and living qualities of the energy we wanted to portray. Our solution was to program the constellations' movements with realistic physics so that they

Still image from the R.E.M. "Animal" music video, 2003. Image courtesy of Motion Theory, Inc.

interacted with Stipe's movements, and seemed to be a real layer of the world. At other times, we created new backgrounds, added insects, and made planetary journeys because we wanted the world of this video to be slightly surreal, as if the frozen moment of the eclipse allows us to see that there is a certain kind of magic out there we can't usually see.

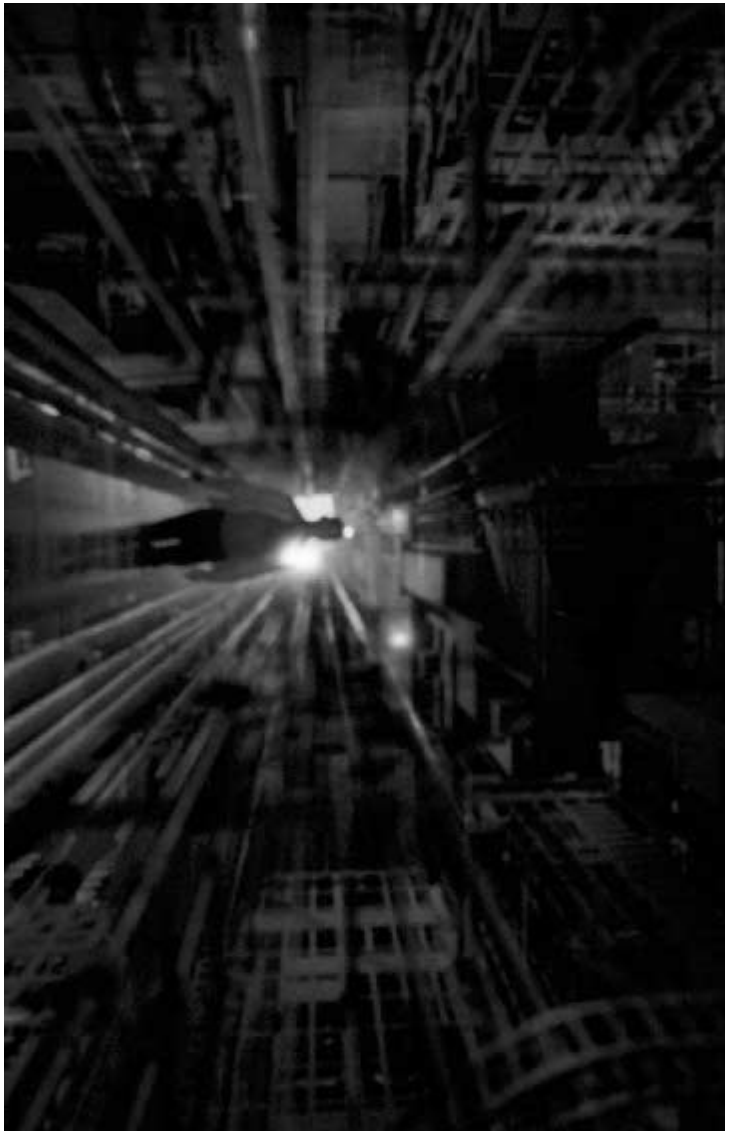
Why do you choose to work with software?

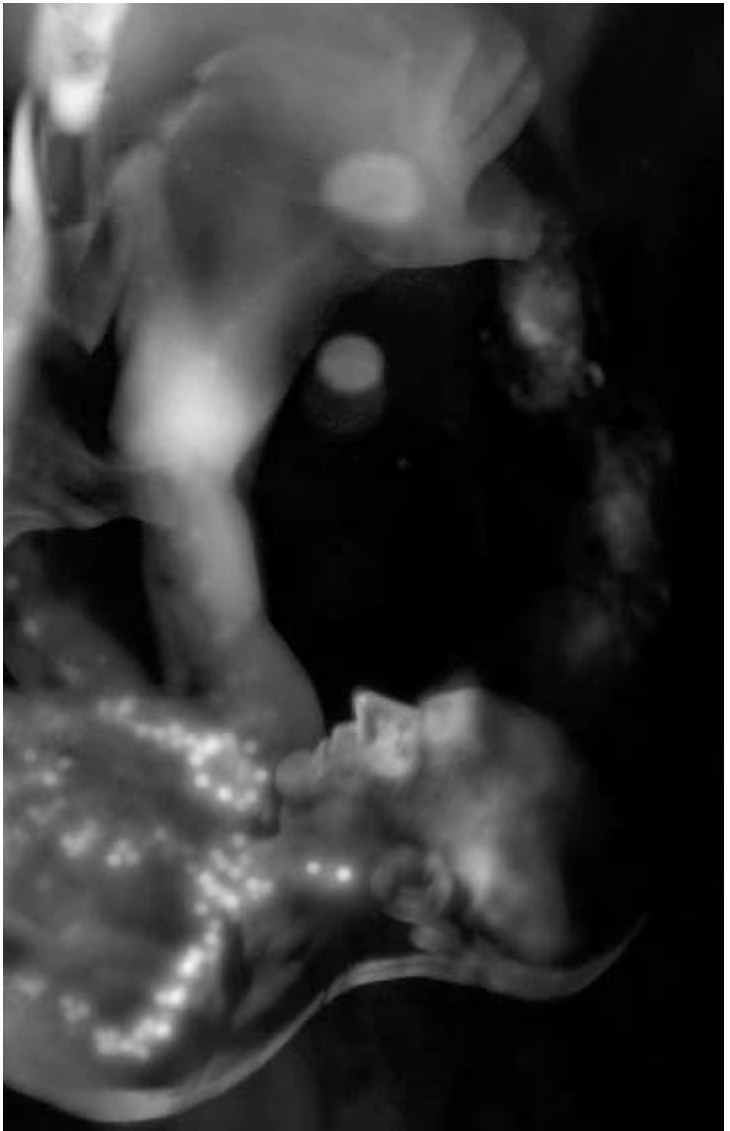
When we started, we didn't really consciously say "We're going to make sure every project we do is different," but when we looked back after a couple of years, we noticed one very important through line in our creative process: the idea comes first. The reason this is important is that we find it's critical not to fall in love with a technique or a tool. To that point, software is just a tool—but it is a tool that serves the imagination well, since it allows us to do things that would otherwise be impossible.

In the case of the R.E.M. video, we were inspired by the themes and scope of the song. We didn't want to settle for anything that we had seen. Our team, including lead programmer Ryan Alexander, spent a lot of R&D time trying to figure out just how to make these different forms of "the energy" look natural and surreal, but not be like a nasty glow effect. For us, this was much more of a philosophical and artistic endeavor than a commercial one, and compromise felt out of the question.

Not every project can be so ambitious, of course, but we are still stubborn enough that we always strive to think of something worth watching and saying—and software—along with, of course, filming, writing, directing, imagination, and luck—empowers us to always find a new way to express it. It seems that software has reached the point where we can basically pour our imaginations out into a frame, or a whole movie, and actually achieve it in a reasonable amount of time. In fact, we're often fortunate enough—and helped by the collaboration of many talented people from many disciplines—to have the result turn out far better than we had first imagined. That's a very strong sign that software is a powerful and vital tool. It's given us this wonderful freedom to just think of good ideas first, because we feel we'll be able to bring them to life in a way that makes the project worth doing.

Still images from the R.E.M. “Animal” music video, 2003. Images courtesy of **Motion Theory, Inc.**





Still images from *Waking Life*. *Waking Life* © 2001 Twentieth Century Fox. All rights reserved.

Waking Life *(Interview with Bob Sabiston)*

Creators	Richard Linklater (writer/director),
	Bob Sabiston (software/animation director)
Year	2001
Medium	Feature film, 35mm
Software	Rotoshop (rotoscoping application written in C++ for Macintosh)
URL	www.wakinglifemovie.com, www.flatblackfilms.com

What is *Waking Life*?

Waking Life *is an animated feature film that takes place entirely within the main character's dream. The film was animated with Rotoshop, a rotoscoping application that produces a floating style of animation well suited to the story. I didn't create the film or have much to do with its content—rather, my contribution is the specific style of animation that* Waking Life *employs and the software used to produce it.*

Why did you create *Waking Life*?

Waking Life *gave me the opportunity to make the type of animated feature film that I would want to watch. I like animation that isn't your garden-variety kid's cartoon, so the chance to do a relatively plotless R-rated philosophical adventure by the director of* Slacker *really was a dream come true.*

What software tools were used?

Rotoshop, the custom rotoscoping software application I have been working on since 1997, was used to animate the film. Thirty artists were hired to manually trace over frames of live-action video footage—interpolation tools in the software allowed them to skip frames of video that didn't change significantly from previous frames. It took roughly a month for an artist to rotoscope one minute of footage. Contrary to many assumptions about the process, there is no element of image processing, filtering, or autotracing in the software. All the software does is interpolate between lines and shapes drawn by the artists—they do all the hard work themselves!

Waking Life *was shot in live action on consumer-level digital videocameras. It was one of the first feature films edited with Final Cut Pro (FCP). Rotoshop uses QuickTime for its source video—the video that you trace—so I exported the finished live-action footage from FCP into five QuickTime reels. At this step the frame rate was reduced to 12 frames per second in order to cut in half the number of frames that artists would have to draw.*

We had sixteen Mac G3 and G4 computers networked, each outfitted with a Wacom graphics tablet. The computers had ten-gigabyte hard drives with a copy of the live-action QuickTime on each machine. Animators traced over this footage to create their animation. We assigned each artist a small section of the film at a time—generally, they were able to work in their own style and weren't required to conform to model sheets or predetermined designs. Finished work was exported across the network to a single computer, where a QuickTime compilation of the completed animation was automatically maintained by the software.

Why did you write your own software tools?

I created the software back in 1996 for an MTV contest. My friends and I had an idea just to animate some real people, documentary subjects that we would go out and interview. I wanted to capture the emotion on people's faces by quickly tracing frame-by-frame in a gestural or life-drawing style. Wrongly I assumed that software was out there to trace over the frames of video—Photoshop could do it, but it was clunky and each frame had to be a separate file. It wasn't quick enough. So, having experience with programming, it seemed like a simple enough thing to write.

The first version took about a week to ten days. Using it I realized that I was drawing the same lines over and over to make a face and that they didn't change a whole lot frame to frame. It was a small step to have the software connect corresponding lines between frames, interpolating to fill in any gaps. Conceptually, this required drawing lines in a certain order from frame to frame, which is not intuitive. You get used to it, though.

By the time Waking Life *came around in 1999, I had used Rotoshop for several animated short films. It had evolved from the simple black-and-white tracing program into more of a full-fledged application. However, the prospect of doing a full feature film inspired a big push to expand the software's capabilities. While Richard Linklater and Sandra Adair were editing the live-action footage, I holed up for a month or two to hammer out improvements in the software. Some of the things added for* Waking Life *were translucency, antialiasing of lines and shapes, hierarchical grouping of layers, and the ability to output to any resolution. I also revamped the user interface for the film's 16 × 9 aspect ratio so that the film frame took up as much of the screen as possible and all the UI controls were pushed into a strip along the bottom of the screen.*

During production, a whole slew of new issues arose simply from having to coordinate and keep track of hundreds of animated scenes. I developed a system for assembling and archiving animation and QuickTime files that worked automatically within the software. As animators finished scenes, they could "publish" them to the movie as a whole so that we always had a watchable version of the movie's current state.

Why do you choose to work with software?

Software has always seemed to me like a variant of the creative process used to write books or paint pictures. It is more constructive and has more limitations, but there is still a great deal of the same inventiveness in play. What's more, when you are finished you have made something that actually does *something. To work on software for use in a larger creative project like a film is even more exciting. There is a satisfaction in building a set of tools that a group of people can use toward pursuit of a common creative goal. It is also very nice to be able to fix problems when they arise and to mold the software to satisfy the needs of the project.*

As someone who enjoys the comparatively less technical activity of animation and rotoscoping, it has been rewarding to be able to bounce back and forth between working on software and then just using the software. The two pursuits play off one another and keep me moving forward.

Still images from *Waking Life*. *Waking Life* © 2001 Twentieth Century Fox. All rights reserved.

Eye Catching, 2003. Photo by Muammer Yanmaz. Image courtesy of ACME, Los Angeles; greengrassi, London; Lehmann Maupin, New York.

Eye Catching *(Interview with Jennifer Steinkamp)*

Creator	Jennifer Steinkamp
Year	2003
Medium	Installation in a sixth-century cistern
Software	Alias Maya, Adobe After Effects, Apple QuickTime, Macromedia Director
URL	www.jsteinkamp.com

What is *Eye Catching*?

Not far from the Hagia Sophia in Istanbul, Turkey, there is a small, inconspicuous brick building where an underground cistern, Yerebatan Sarnici*—the Sunken Palace—was built in the sixth century by order of Emperor Justinian as a reservoir for the Great Palace of the Byzantine Empire. The ceiling of the enormous cistern is supported by 336 columns salvaged from other sites—Doric, Ionic, and Corinthian styles; a wooden boardwalk allows visitors to explore the mysterious space. Eye Catching is a computer video-projected installation within the cistern. The title "Eye Catching" is a play on words about Medusa.*

Why did you create *Eye Catching*?

I was invited to create a work of art for the Eighth Istanbul Biennial, 2003. I visited Istanbul six months before the exhibition to survey the various sites. The curator, Dan Cameron, suggested my art would work well in the Yerebatan cistern. (Curators always seem to know where my art will work.) I noticed the ancient Medusa heads used as column supports. I thought this would be a good area to project animation. I had no idea what I would do, especially since I did not really remember Medusa's story—except for the 1981 film Clash of the Titans.

Typically for an art installation, I will take measurements of the site, and then create a 3D model in Maya software. I use this model to calculate the projector placement and determine the motion. The site was so old and full of water, it was impossible to assess; because of this constraint, I set up a studio with three computers in my hotel room one week before the exhibition to make the final renderings of the piece. The same computers were also used in the exhibition.

While I was stressing over what I could possibly do in this complicated site, I researched the incredible story of Medusa, realizing that there was a feminist psychological interpretation of the tale. Medusa was an extraordinarily beautiful woman; a sea god raped her. (There are two different versions of who actually raped Medusa.) Then, because of jealousy, she was transformed by the goddess Athena into an incredible monster with serpentine hair and a gaze that would turn men to stone. One interpretation could be that this was the ultimate extension of the power of female sexuality (stone as erection), and the fear and paranoia this can invoke in men. I created serpentine trees to add to the enchanted environment of the cistern, as if Medusa's sensuality transformed the environment and everything around her. One of the trees was old, with no leaves; it was created to seem dead, brought back to life.

What software tools were used?

I used Alias Maya Paint Effects to create and animate the trees. Adobe Photoshop was used to make paintings of the leaves, flowers, and bark used as texture maps. Adobe After Effects was used to composite the animation loops and render out to Apple QuickTime, Sorenson

compressed 1024 × 768 movies. Macromedia Director was used to oscillate the QuickTime movies and randomly vary the playback rates, loading the movies into one gigabyte of RAM.

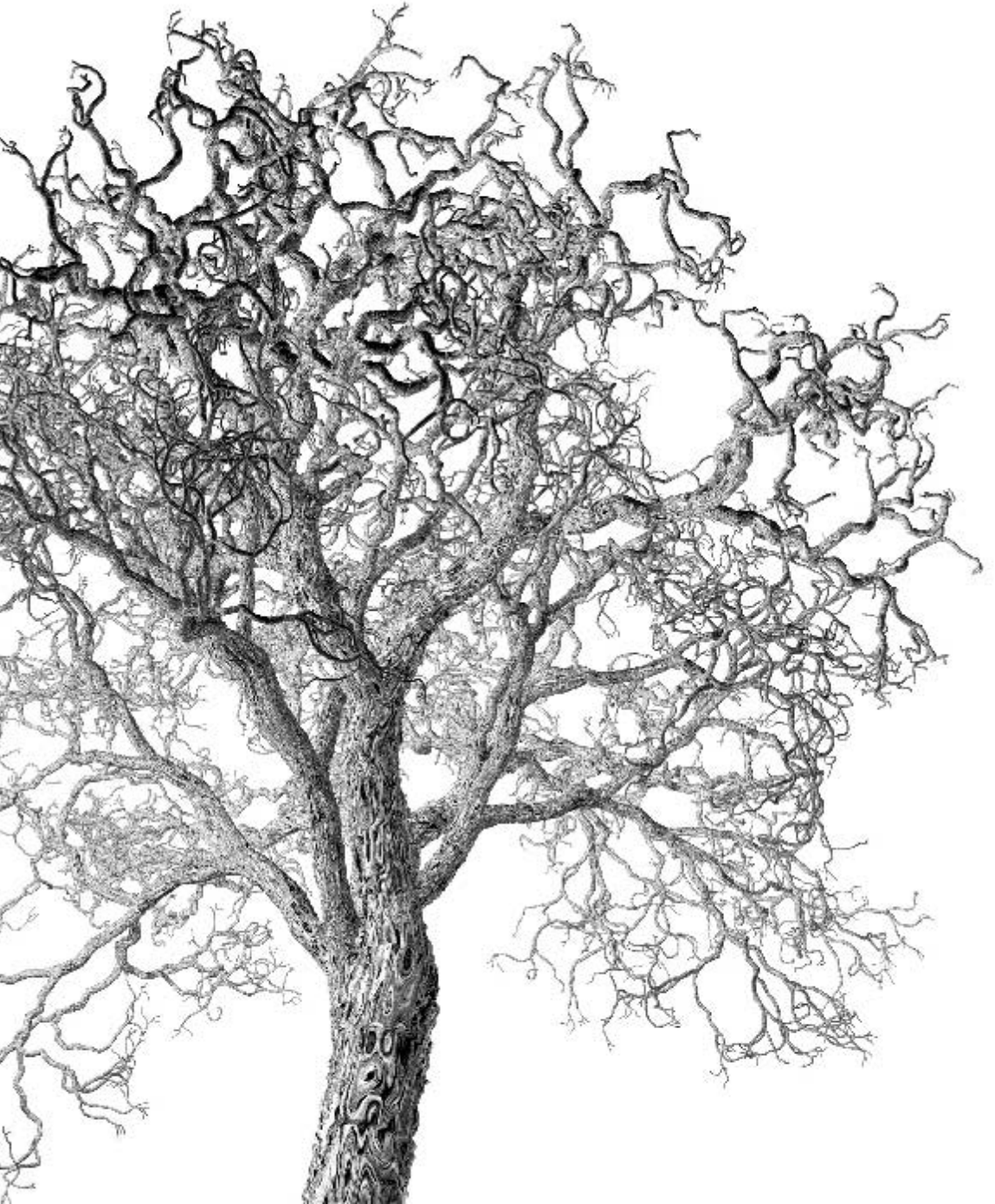
Why did you use these tools?

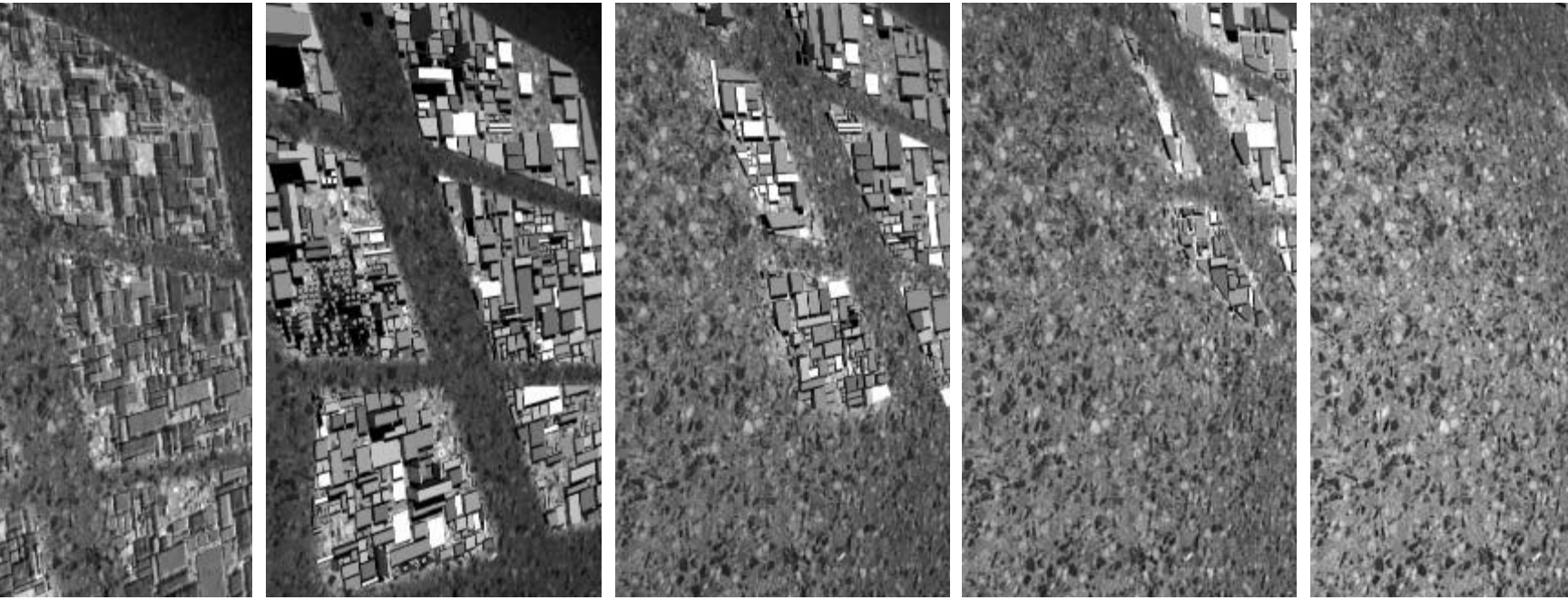
I use Maya and Photoshop because they are very deep, capable programs—there is a lot of unexplored territory there. One could spend years exploring a multitude of artistic ideas. Director is a pretty fast, handy method to control QuickTime. I can use Director to make a stand-alone program to run on a PC and automatically start when the PC powers on. This is good for the collectors, gallery and museum people who know little about computers.

Why do you choose to work with software?

I have been using computer software since 1982. I first came across it when I enrolled in Gene Youngblood's video art course at Caltech where I saw some of the first computer animation artwork. I saw computer graphics as a means to explore new ideas, images, and motion. I still possess this excitement. I use 3D software to create virtual objects and space and I then place these into real space. Both the real and the virtual spaces are transformed by each other—forming an in-between space, a space your body can understand.

Still image from *Eye Catching*, 2003. Image courtesy of ACME, Los Angeles; greengrassi, London; Lehmann Maupin, New York.





The Mini-Epoch Series, 2003. Images courtesy of the artists.

The Mini-Epoch Series *(Interview with Semiconductor)*

Creators	Semiconductor (Ruth Jarman and Joseph Gerhardt)
Year	2003
Medium	Installation
Software	Adobe Premiere, Autodesk 3ds Max, Adobe Photoshop
URL	www.semiconductorfilms.com

What is *The Mini-Epoch Series*?

The Mini-Epoch Series *installation is five one-minute sound films, each installed on a 7" widescreen LCD display. The films were shot, animated, and exhibited in Palazzo Zenobio, Venice.*

Screen One: Stop-frame animation of a puddle drying up over a period of four hours in Palazzo Zenobio courtyard. Composited graphics represent population density fluctuating over hundreds of years according to the availability of water, as the lake depletes. The sound controls the statistics according to the rate of evaporation.

Screen Two: Stop-frame animation of the sun moving across the floor of Palazzo Zenobio. Animated within the sun's path are composited graphical representations of land use, which adapt to the availability of sunlight over thousands of years. The sound fluctuation is consistent with the strength of sunlight, which in turn controls the transition of land use.

Screen Three: Fictional animation of the sun moving across a Palazzo Zenobio room and up the wall; made using actual data of the sun's path for that time and place. We track the window's path across the wall as buildings are constructed in the city, forming animated silhouettes. The shadow from the encroaching city subsequently blocks out the light and sends the room into total darkness. Here we witness the construction of Venice's past or future over hundreds of years.

Screen Four: Stop-frame animation of a peeling painted wall within the exhibition space. As the landscape shifts over thousands of years the inhabitants migrate across this desertscape.

Screen Five: Animation bringing to life the motion of the Venetian architecture in the saturated terrain over hundreds of years.

Why did you create *The Mini-Epoch Series*?

We were invited to create a site-specific artwork for the Venice Biennale 2003, in Palazzo Zenobio, Venice. Many layers of topography surround the exhibition location. The historic Palazzo and its courtyard are encircled by the famous canals of Venice; these again are enclosed by a lagoon that is also bounded by the sea. No city we have ever been to has had such a clear ecological destiny. The city was founded by a unique collection of nomads and travelers who wished to stay away from the permanence of solid ground but who are now long gone, leaving this symbolic and crumbling dead city, or at best a living museum. The temporality is apparent all around, making evident to us the fate of all cities and eventually all places we know. This humbling but sentimental view is withheld from the work. Instead we associate with the architecture's own impartial view on the matter. Each element of the work, which spanned five screens, portrays a microtopology and its statistical changes over time. Reanimating some facet of a fictional civilization's progress and ultimate demise, we reveal perspectives that are invisible to us as a result of our short life span.

What software tools were used?

We used Adobe Premiere for time-lapse animation, 3ds Max and Max script for modeling and animating according to the sound, and Adobe Photoshop for image editing.

Why did you use these tools?

These tools gave us access to functions required to animate, construct, and composite the work. The initial work was done with stop-motion/time-lapse capturing in Adobe Premiere (a function removed in recent versions). This allowed us to capture a time lapse animation of environmental changes, e.g., the light changing as shadows move through the space or a puddle drying up in the sun after a passing shower. Within 3ds Max we composited, scripted, and synchronized the sound and image.

Why did you write your own software tools?

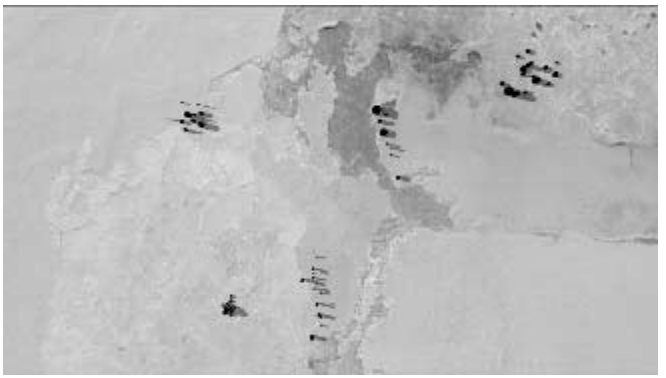
An artist's dependence on globally used software tends to homogenize the output. Styles quickly become very recognizable and thus diminish the ability to challenge the artist or the viewer. When we're creating work we always make explicit demands on the software, rather than letting it dictate the aesthetic or the direction of the work. This often means we are driven to create our own scripts, code, and more recently our own real-time performance software, Sonic Inc—a real-time drawing/sculpting tool, which builds by, and responds to, sound. On Screen Three of Mini-Epochs*, we implemented a script we had initially written for our Sound Film* Inaudible Cities: Part One*. It listens to the sound and generates buildings accordingly, creating a whole metropolis block by block. This script was used to create the construction of buildings beyond the window. On other screens, programming was applied to control the relationship between the sound and the animation of data in the time-lapse environments.*

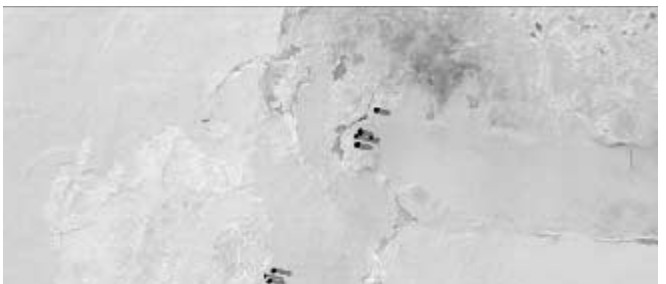
Why do you choose to work with software?

The advent of the domestic computer presented us with a new artistic medium to probe and enabled us to explore relationships between sound and image in the digital domain. Through our creation of Sound Films we began to reveal our physical world in flux: cities in motion, shifting landscapes, and systems in chaos. Central to these works is the role of sound, which becomes synonymous with the image as it creates, controls, and deciphers it; exploring resonance through the natural order of things. Working within the realm of the computer and utilizing software created for the film and game industries, we are able to manipulate our physical environment, controlling events on a macro and micro scale while testing the technology's potential, stretching it and sculpting it.

Computer software is made for a specific function and consequently has a distinct identity, whether this is controlling the "realistic" look of a 3D object or defaulting a set of parameters to animate a particular motion. This poses a constant struggle for the creative user, as the computer tries to impose its own signature on our work. We began to incorporate these nuances as part of the making process, often subverting the software's intended use by finding a new path in which to resculpt the data, or forging new associations between digital and analog. In several works we assigned specific responsibilities to the computer, encouraging its participation. This identifies with our proposal of Artificial Expressionism, a pledge between the computer and the artist: the artificial, indicative of zeros and ones, combined with the human expression, the unpredictable element. Our artistic name Semiconductor also acknowledges the computer as co-conspirator; half conducting the art work and forming a relationship with us the artist.

The Mini-Epoch Series, 2003. Images courtesy of the artists.





Structure 4: Objects I

This unit introduces the concept of object-oriented programming and presents the code elements for working with objects.

Syntax introduced:
`class, Object`

Variables and functions are the building blocks of software. Several functions will often be used together to work on a set of related variables. Object-oriented programming, which was developed to make this process more explicit, uses objects and classes as building blocks. A class defines a group of methods (functions) and fields (variables). An object is a single instance of a class. The fields within an object are typically accessible only via its own methods, allowing an object to hide its complexity from other parts of a program. This resembles interfaces built for other complex technologies; the driver of a car does not see the complexity of the engine while in motion, although the speed and RPM are readily visible on the console. The same type of abstraction is used in object-oriented programming to make code easier to understand and reuse in other contexts.

Object-oriented programming is a different way of thinking about programming, but it builds on the previously introduced concepts. The technology for object-oriented programming existed long before the practice became popular in the mid-1980s and gradually went on to become the dominant way to think about software. Many people find it to be a more intuitive way to think about programming. In addition to providing a helpful conceptual model, object-oriented programming becomes a necessity when a program includes many elements or when it grows larger than a few pages of code. Objects can provide a powerful way to think about structuring your ideas in code, and you'll find a number of examples with objects throughout the rest of this text.

All software written in Processing consists of objects, but this fact is initially hidden so that object-oriented programming concepts can be introduced later. Unless code is made explicitly object-oriented, clicking the Run button transparently adds extra syntax that wraps a sketch as an object.

Object-oriented programming

A modular program is composed of code modules that each perform a specific task. Variables are the most basic way to think about reusing elements within a program. They allow a single value to appear many times within a program and to be easily changed. Functions abstract a specific task and allow code blocks to be reused throughout a program. Typically, one is concerned only with what a function does, not how it works. This frees the mind to focus on the goals of the program rather than on

the complexities of infrastructure. Object-oriented programming further extends the modularity of using variables and writing functions by allowing related functions to be grouped together.

It's possible to make an analogy between software objects and real-world artifacts. To get you in the spirit of thinking about the world through the object-oriented lens, we've created a list of everyday items and a few potential fields and methods for each.

```
Name      Apple
Fields    color, weight
Methods   grow(), fall(), rot()

Name      Butterfly
Fields    species, gender
Methods   flapWings(), land()

Name      Radio
Fields    frequency, volume
Methods   turnOn(), tune(), setVolume()

Name      Car
Fields    make, model, color, year
Methods   accelerate(), brake(), turn()
```

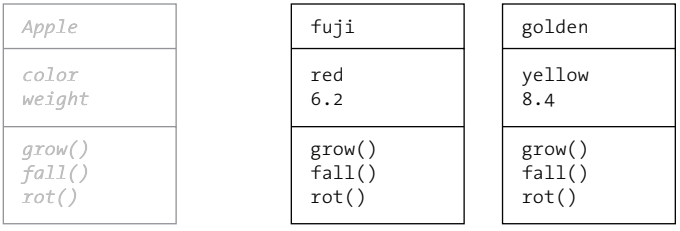
Extending the apple example reveals more about the process of thinking about the world in relation to software objects. To make a software simulation of the apple, the `grow()` method might have inputs for temperature and moisture. The `grow()` method can increase the weight field of the apple based on these inputs. The `fall()` method can continually check the weight and cause the apple to fall to the ground when the weight goes above a threshold. The `rot()` method could then take over, beginning to decrease the value of the weight field and change the color fields.

As explained in the introduction, objects are created from a class and a class describes a set of fields and methods. An instance of a class is a variable, and like other variables, it must have a unique name. If more than one object is created from a class, each must have a unique name. For example, if two objects named `fuji` and `golden` are created using the `Apple` class, each can have its own values for its fields:

```
Name      fuji
Fields    color: red
          weight: 6.2

Name      golden
Fields    color: yellow
          weight: 8.4
```

Two popular styles of class diagrams are tables and a circular format inspired by biological cells. Each diagram style shows the name of the class, the fields, and the methods. It is useful to diagram classes in this way to define their characteristics before starting to code. The diagrams are also useful because they show the components of a

class without including too much detail. Looking at the `Apple` class and the `fuji` and `golden` objects created from it, you can see how these diagrams work:



Apple class fuji object golden object

The circular diagrams reinforce *encapsulation*, the idea that an object's fields should not be accessible from the outside. The methods of an object should act as a buffer between code outside the class and the data contained within:

Apple class fuji object golden object

The fields and methods of an object are accessed with the dot operator, a period. To get the color value from the `fuji` object, the syntax `fuji.color` accesses the value of the color field inside the `fuji` object. The syntax `golden.color` accesses the value of the color field inside the golden object. The dot operator is also used to activate (or "call") the methods of the object. To run the `grow()` method inside the golden object, the syntax `golden.grow()` is used.

With the concepts and terminology discussed in this unit (object, class, field, method, encapsulation, and dot operator), you are equipped to begin the journey into object-oriented programming, which is explained further in Structure 5 (p. 453).

Using classes and objects

Defining a class is creating your own data type. Unlike the primitive types `int`, `float`, and `boolean`, it's a composite type like `String`, `PImage`, and `PFont`, which means it can hold many variables and methods inside one name. When creating a class, first think carefully about what you want the code to do. It's common to make a list of variables required (these will be the fields) and figure out what type they should be.

The code on the following pages creates the same image of a white dot on the black background, but the code is written in different ways. In the first example program, a circle is positioned on screen. It needs two fields to store the location. These variables are `float` values that will provide more flexibility to control the circle's movement. The circle also needs a size, so we've created the `diameter` field to store its diameter:

`float x`	*X-coordinate of the circle*
`float y`	*Y-coordinate of the circle*
`float diameter`	*Diameter of the circle*

The name of a class should be carefully considered. The name can be nearly any word, adhering to the same naming conventions as variables (p. 40); however, class names should always be capitalized. This helps separate a class like `String` or `PImage` from the lowercase names of primitive types like `int` or `boolean`. The name `Spot` was chosen for this example because a spot is drawn to the screen (the name "Circle" also would have made sense). As with variables, it can be very helpful to give a class a name that matches its purpose.

Once the fields and name for the class definition have been determined, consider how the program would be written without the use of an object. In the following example, the data for the ellipse's position and diameter is a part of the main program. In this case, that's not a problem, but the use of several ellipses or complex motion would make the program unwieldy.

43-01

```
float x = 33;
float y = 50;
float diameter = 30;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  background(0);
  ellipse(x, y, diameter, diameter);
}
```

To make this code more generally useful, the next example moves the fields that pertain to the ellipse into their own class. The first line in the program declares the object `sp` of the type `Spot`. The `Spot` class is defined after the `setup()` and `draw()`. The `sp` object is constructed within `setup()`, after which its fields can be accessed and assigned. The next three lines assign values to the fields within `Spot`. These values are accessed inside `draw()` to set the position and size of the ellipse. The dot operator is used to assign and access the variables within the class.

```
Spot sp;              // Declare the object                    43-02

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot();    // Construct the object
  sp.x = 33;          // Assign 33 to the x field
  sp.y = 50;          // Assign 50 to the y field
  sp.diameter = 30;   // Assign 30 to the diameter field
}

void draw() {
  background(0);
  ellipse(sp.x, sp.y, sp.diameter, sp.diameter);
}

class Spot {
  float x, y;         // The x- and y-coordinate
  float diameter;     // Diameter of the circle
}
```

The `Spot` class as it exists is not very useful, but it's a start. This next example builds on the previous one by adding a method to the `Spot` class—this is one more step toward using object-oriented programming to its advantage. The `display()` method has been added to the class definition to draw the element to the screen:

`void display()` ***Draws the spot to the display window***

In the code below, the last line inside `draw()` runs the `display()` method for the `sp` object by writing the name of the object and the name of the method connected with the dot operator. Also notice the difference in the parameters of the ellipse function in code 43-02 and 43-03. In code 43-03, the name of the object is not used to access the `x`, `y`, and `diameter` fields. This is because the `ellipse()` function is called from within the `Spot` object. Because this line is a part of the object's `display()` function, it can access its own variables without specifying its own name.

It's important to reinforce the difference between the `Spot` class and the `sp` object in this example. Although the code might make it look like the fields `x`, `y`, and `diameter` and the method `display()` belong to `Spot`, this is just the definition for any object created from this class. Each of these elements belong to (are encapsulated by) the `sp` variable, which is one instance of the Spot data type.

43-03

```
Spot sp;            // Declare the object

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot();  // Construct the object
  sp.x = 33;
  sp.y = 50;
  sp.diameter = 30;
}

void draw() {
  background(0);
  sp.display();
}

class Spot {
  float x, y, diameter;

  void display() {
    ellipse(x, y, diameter, diameter);
  }
}
```

The next example introduces a new programming element called a *constructor*. A constructor is a block of code activated as the object is created. The constructor always has the same name as the class and is typically used to assign values to an object's fields as it comes into existence. (If there is no constructor, the value of every numeric field is set to zero.) The constructor is like other methods except that it is not preceded with a data type or the keyword `void` because there is no return type. When the object `sp` is created, the parameters 33, 50, and 30 are assigned in order to the variables `xpos`, `ypos`, and `dia` within the constructor. Within the constructor block, these values are assigned to the object's fields `x`, `y`, and `diameter`. For the fields to be accessible within every method of the object, they must be declared outside of the constructor. Remember the rules of variable scope (p. 178)—if the fields are declared within the constructor, they cannot be accessed outside the constructor.

43-04

```
Spot sp;                        // Declare the object

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot(33, 50, 30);  // Construct the object
}

void draw() {
  background(0);
  sp.display();
}

class Spot {
  float x, y, diameter;

  Spot(float xpos, float ypos, float dia) {
    x = xpos;        // Assign 33 to x
    y = ypos;        // Assign 50 to y
    diameter = dia;  // Assign 30 to diameter
  }

  void display() {
    ellipse(x, y, diameter, diameter);
  }
}
```

The behavior of the `Spot` class can be extended by the addition of more methods and fields to the definition. The following example extends the class so that the ellipse moves up and down the display window and changes direction when it collides with the top or bottom. Since the class will be moving, it needs a field to set the speed, and because it will change directions, it needs a field to hold the current direction. We've named these fields `speed` and `direction` to make their uses clear and the names short. We decided to make the `speed` a `float` value to give a broader range of possible speeds. The `direction` field is an `int` so that it can be easily incorporated into the math for its movement:

`float speed`	***Distance moved each frame***
`int direction`	***Direction of motion (1 is down, -1 is up)***

To create the desired motion, we need to update the position of the circle on each frame. The direction also has to change at the edges of the display window. To test for an edge, the code tests whether the y-coordinate is smaller than the circle's radius or larger than

the height of the window minus the circle's radius. Make sure to include the radius value; then the direction will change when the outer edge of the circle (rather than its center) reaches the edge. In addition to deciding what the methods need to do and what they should be called, we must also consider the return type. Because nothing is returned from this method, the keyword `void` is used:

`void move()`	***Updates the circle's position and direction***

The code within the `move()` and `display()` methods could have been combined in one method; they were separated to make the example more clear. Changing the position of the object is a separate task from drawing it to the screen, and using separate methods reflects this. These changes allow every object created from the `Spot` class to have its own size and position. The objects will also move up and down the screen, changing directions at the edge.

```
class Spot {
  float x, y;         // X-coordinate, y-coordinate
  float diameter;     // Diameter of the circle
  float speed;        // Distance moved each frame                    43-05
  int direction = 1;  // Direction of motion (1 is down, -1 is up)

  // Constructor
  Spot(float xpos, float ypos, float dia, float sp) {
    x = xpos;
    y = ypos;
    diameter = dia;
    speed = sp;
  }

  void move() {
    y += (speed * direction);
    if ((y > (height - diameter/2)) || (y < diameter/2)) {
      direction *= -1;
    }
  }

  void display() {
    ellipse(x, y, diameter, diameter);
  }
}
```

To save space and to keep the focus on the reuse of objects, examples from here to the end of the unit won't reprint the code for the `Spot` class in examples that require it. Instead, when you see a comment like `// Insert Spot class`, cut and paste the code

for the class into this position to make the code work. Run the following code to see the result of the `move()` method updating the fields and the `display()` method drawing the `sp` object to the display window.

43-06

```
Spot sp; // Declare the object

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp = new Spot(33, 50, 30, 1.5);  // Construct the object
}

void draw() {
  fill(0, 15);
  rect(0, 0, width, height);
  fill(255);
  sp.move();
  sp.display();
}

// Insert Spot class
```

Like a function, a well-written class enables the programmer to focus on the resulting behavior and not the details of execution. Objects should be built for the purpose of reuse. After a difficult programming problem is solved and encoded inside an object, that code can be used later as a tool for building new code. For example, the functions and classes used in Processing grew out of many commonly used functions and classes that were part of the authors' own code.

As long as the interface to the class remains the same, the code within can be updated and modified without breaking a program that uses the object. For example, as long as the object is constructed with the x-coordinate, y-coordinate, and diameter and the names of `move()` and `display()` remain the same, the actual code inside `Spot` can be changed. This allows the programmer to refine the code for each object independently from the entire program.

Like other types of variables, additional objects are added to a program by declaring more names. The following program has three objects made from the `Spot` class. These objects, named `sp1`, `sp2`, and `sp3`, each have their own set of fields and methods. A method for each object is run by specifying its name, followed by the dot operator and the method name. For example, the code `sp1.move()` runs the `move()` method, which is a part of the `sp1` object. When these methods are run, they access the fields within their object. When `sp3` runs `move()` for the first time, the field value `y` is updated by the `speed` field value of 2.0 because that value was passed into `sp3` through the constructor.

43-07

```
Spot sp1, sp2, sp3;  // Declare the objects

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  sp1 = new Spot(20, 50, 40, 0.5);  // Construct sp1
  sp2 = new Spot(50, 50, 10, 2.0);  // Construct sp2
  sp3 = new Spot(80, 50, 30, 1.5);  // Construct sp3
}

void draw() {
  fill(0, 15);
  rect(0, 0, width, height);
  fill(255);
  sp1.move();
  sp2.move();
  sp3.move();
  sp1.display();
  sp2.display();
  sp3.display();
}

// Insert Spot class
```

It's difficult to summarize the basic concepts and syntax of object-oriented programming using only one example. To make the process of creating objects easier to comprehend, we've created the `Egg` class to compare and contrast with `Spot`. The `Egg` class is built with the goal of drawing an egg shape to the screen and wobbling it left and right. The `Egg` class began as an outline of the fields and methods it needed to have the desired shape and behavior:

`float x`	***X-coordinate for middle of the egg***
`float y`	***Y-coordinate for bottom of the egg***
`float tilt`	***Left and right angle offset***
`float angle`	***Used to define the tilt***
`float scalar`	***Height of the egg***
`void wobble()`	***Moves the egg back and forth***
`void display()`	***Draws the egg***

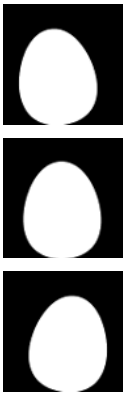
After the class requirements were established, it developed the same way as the `Spot` class. The `Egg` class started minimally, with only `x` and `y` fields and a `display()` method. The class was then added to a program with `setup()` and `draw()` to check the result. The `scale()` function was added to `display()` to decrease the size of the egg.

When this first program was working to our satisfaction, the `rotate()` method and `tilt` field were added to change the angle. Finally, the code was written to make the egg move. The `angle` field was added as a continuously changing number to set the tilt. The `wobble()` method was added to increment the angle and calculate the tilt. The `cos()` function was used to accelerate and decelerate the wobbling from side to side. After many rounds of incremental additions and testing, the final `Egg` class was working as initially planned.

43-08

```
class Egg {
  float x, y;    // X-coordinate, y-coordinate
  float tilt;    // Left and right angle offset
  float angle;   // Used to define the tilt
  float scalar;  // Height of the egg

  // Constructor
  Egg(int xpos, int ypos, float t, float s) {
    x = xpos;
    y = ypos;
    tilt = t;
    scalar = s / 100.0;
  }

  void wobble() {
   tilt = cos(angle) / 8;
   angle += 0.1;
  }

  void display() {
    noStroke();
    fill(255);
    pushMatrix();
    translate(x, y);
    rotate(tilt);
    scale(scalar);
    beginShape();
    vertex(0, -100);
    bezierVertex(25, -100, 40, -65, 40, -40);
    bezierVertex(40, -15, 25, 0, 0, 0);
    bezierVertex(-25, 0, -40, -15, -40, -40);
    bezierVertex(-40, -65, -25, -100, 0, -100);
    endShape();
    popMatrix();
  }
}
```

The Egg class is included in setup() and draw() the same way as in the Spot examples. An object of type Egg called humpty is created outside of setup() and draw(). Within setup(), the humpty object is constructed and the coordinates and initial tilt value are passed to the constructor. Within draw(), the wobble() and display() functions are run in sequence, causing the egg's angle and tilt values to update. These values are used to draw the shape to the screen. Run this code to see the egg wobble from left to right.



43-09

```
Egg humpty;  // Declare the object

void setup() {
  size(100, 100);
  smooth();
  // Inputs: x-coordinate, y-coordinate, tilt, height
  humpty = new Egg(50, 100, PI/32, 80);
}

void draw() {
  background(0);
  humpty.wobble();
  humpty.display();
}

// Insert Egg class
```

The Spot and Egg classes are two simple objects used to convey the basic syntax and concepts involved in object-oriented programming.

Arrays of objects

Working with arrays of objects is similar to working with arrays of other data types. Like all arrays, an array of objects is distinguished from a single object with brackets, the [and] characters. Because each array element is an object, each element of the array must be created before it can be accessed. The steps for working with an array of objects are:

1. Declare the array
2. Create the array
3. Create each object in the array

These steps are translated into code in the following example:

```
int numSpots = 6;
// Declare and create the array
Spot[] spots = new Spot[numSpots];

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  for (int i = 0; i < spots.length; i++) {
    float x = 10 + i*16;
    float rate = 0.5 + i*0.05;
    // Create each object
    spots[i] = new Spot(x, 50, 16, rate);
  }
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  for (int i = 0; i < spots.length; i++) {
    spots[i].move();     // Move each object
    spots[i].display();  // Display each object
  }
}

// Insert Spot class
```

The `Ring` class presents another example of working with arrays and objects. This class defines a circle that can be turned on, at which point it expands to a width of 400 and then stops displaying to the screen by turning itself off. When this class is added to the example below, a new ring turns on each time a mouse button is pressed. The fields and methods for `Ring` make this behavior possible:

`float x`	***X-coordinate of the ring***
`float y`	***Y-coordinate of the ring***
`float diameter`	***Diameter of the ring***
`boolean on`	***Turns the display on and off***
`void grow()`	***Increases the diameter if on is true***
`void display()`	***Draws the ring***

`Ring` was first developed as a simple class. Its features emerged through a series of iterations. This class has no constructor because its values are not set until the `start()` method is called within the program.

```
class Ring {
  float x, y;          // X-coordinate, y-coordinate
  float diameter;      // Diameter of the ring
  boolean on = false;  // Turns the display on and off

  void start(float xpos, float ypos) {
    x = xpos;
    y = ypos;
    on = true;
    diameter = 1;
  }

  void grow() {
    if (on == true) {
      diameter += 0.5;
      if (diameter > 400) {
        on = false;
      }
    }
  }

  void display() {
    if (on == true) {
      noFill();
      strokeWeight(4);
      stroke(155, 153);
      ellipse(x, y, diameter, diameter);
    }
  }
}
```

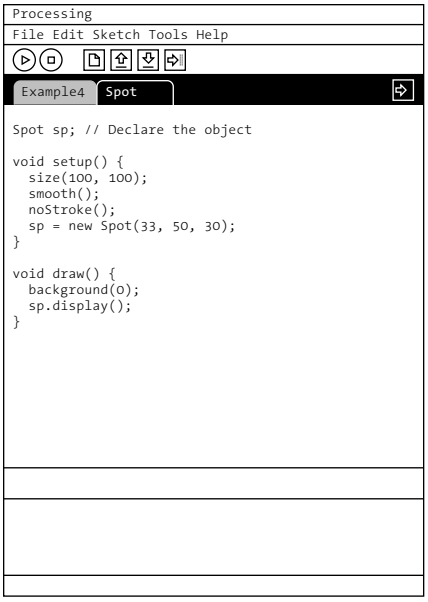
In this program, the `rings[]` array is created to hold fifty `Ring` objects. Space in memory for the `rings[]` array and `Ring` objects is allocated in `setup()`. The first time a mouse button is pressed, the first Ring object is turned on and its `x` and `y` variables are assigned the current values of the cursor. The counter variable `currentRing` is incremented by one, so the next time through the `draw()`, the `grow()` and `display()` methods will be run for the first `Ring` element. Each time a mouse button is pressed, a new `Ring` is turned on and displayed in the subsequent trip through `draw()`. When the final element in the array has been created, the program jumps back to the beginning of the array to assign new positions to earlier `Rings`.

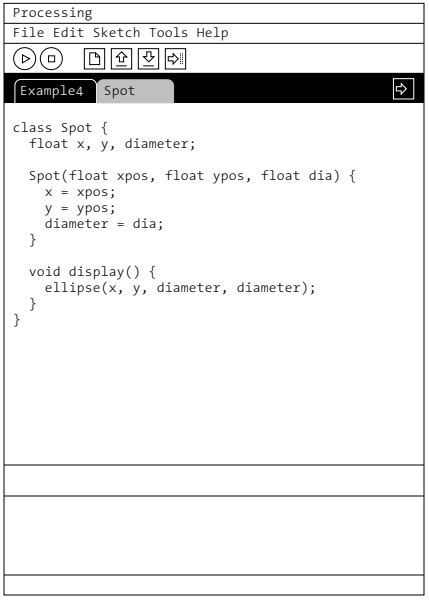
```
Ring[] rings;                   // Declare the array
int numRings = 50;
int currentRing = 0;

void setup() {
  size(100, 100);
  smooth();
  rings = new Ring[numRings];  // Create the array
  for (int i = 0; i < numRings; i++) {
    rings[i] = new Ring();     // Create each object
  }
}

void draw() {
  background(0);
  for (int i = 0; i < numRings; i++) {
    rings[i].grow();
    rings[i].display();
  }
}

// Click to create a new Ring
void mousePressed() {
  rings[currentRing].start(mouseX, mouseY);
  currentRing++;
  if (currentRing >= numRings) {
    currentRing = 0;
  }
}

// Insert Ring class
```

As modular code units, objects can be utilized in diverse ways according to the desires of different people for the needs of different projects. This is one of the exciting things about programming with objects.

Multiple files

The programs written before this unit have used one file for all of their code. As programs become longer, a single file can become inconvenient. When programs grow to hundreds and thousands of lines, breaking programs into modular units helps manage different parts of the program. Processing manages files with the Sketchbook, and each sketch can have multiple files that are managed with tabs.





Multiple Files

Programs can be divided into different files and represented as tabs within the PDE. This makes it easier to manage complicated programs.

The arrow button in the upper-right corner of the Processing Development Environment (PDE) is used to manage these files. Clicking this button reveals options to create a new tab, rename the current tab, and delete the current tab. If a project has more than one tab, each tab can also be hidden and revealed with this button. Hiding a tab temporarily removes that code from the sketch.

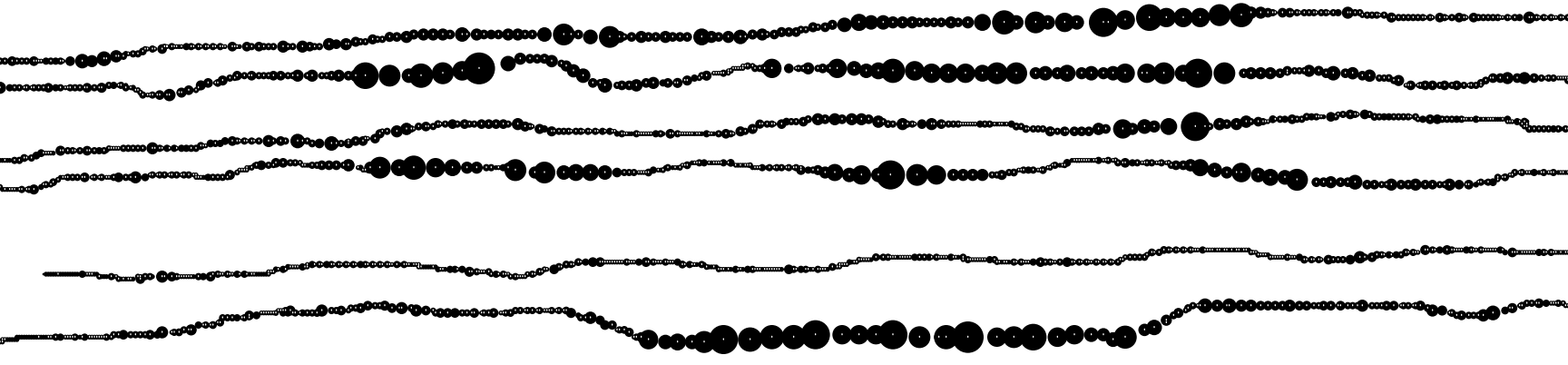
Code 43-04 can be divided into separate files to make it into a more modular program. First open or retype this program into Processing and name it "Example4." Now click on the arrow button and select the New Tab option from the menu that appears. A prompt asking for the name of the new tab appears. Type the name you want to assign to the file and click "OK" to continue. Because we'll be storing the `Spot` class in this file, use the name "Spot." You now have a new file called *Spot.pde* in your sketch folder. Select "Show Sketch Folder" from the Sketch menu to see this file.

Next, click on the original tab and and select the text for the `Spot` class. Cut the text from this tab, change to the `Spot` tab, and paste. Save the sketch for good measure, and press the Run button to see how the two files combine to create the final program. The file *Spot.pde* can be added to any sketch folder to make the `Spot` class accessible for that sketch.

When a sketch is created with multiple files, one of the files must have the same name as the folder containing the sketch to be recognized by Processing. This file is the main file for the sketch and always appears as the leftmost tab. The `setup()` and `draw()` methods for the sketch should be in this file. Only one of the files in a sketch can have a `setup()` and `draw()`. The other tabs appear in alphabetical order from left to right. When a sketch is run, all of the PDE files that comprise a sketch are converted to one file before the code is compiled and run. Additional functions and variables in the additional tabs have access to all global variables in the main file with `setup()` and `draw()`. Advanced programmers may want a different behavior, and a more detailed explanation can be found in the reference.

Exercises

1. *Write your own unique* `Spot` *class that has a different behavior than the one presented in the example. Design a kinetic composition with 90 of your* `Spots`.
2. *Design a class that displays, animates, and defines the behavior of an organism in relation to another object made from the same class.*
3. *Create a class to define a software puppet that responds to the mouse.*



Drawing 2: Kinetic Forms

This unit focuses on developing kinetic drawing tools and elements unique to software.

The experimental animation techniques of drawing, painting, and scratching directly onto film are all predecessors to software-based kinetic drawings. The immediacy and freshness of short films such as Norman McLaren's *Hen Hop* (1942), Len Lye's *Free Radicals* (1957), and Stan Brakhage's *The Garden of Earthly Delights* (1981) is due to the extraordinary qualities of physical gesture which software later made more accessible. In his 1948 essay "Animated Films," McLaren wrote, "In one operation, which is drawing directly onto the 35mm clear machine leader with an ordinary pen nib and India ink, a clean jump was made from the ideas in my head to the images on what would normally be called a developed negative." He further explains, "The equivalents of Scripting, Drawing, Animating, Shooting, Developing the Negative, Positive Cutting, and Negative Cutting were all done in one operation."[1] Like working directly on film, programming provides the ability to produce kinetic forms with immediate feedback.

Software animation tools further extend film techniques by allowing the artist to edit and animate elements continuously after they have been drawn. In 1991, Scott Snibbe's *Motion Sketch* extended to software the techniques explored by McLaren, Lye, and Brakhage. The application translates hand motion to visual elements on the screen. Each gesture creates a shape that moves in a one-second loop. The resulting animations can be layered to create a work of spatial and temporal complexity reminiscent of Oskar Fischinger's style. Snibbe extended this concept further with *Motion Phone* (1995), which enabled people to work simultaneously in a shared drawing space via the Internet.

Many artists have developed their own software in pursuit of creative animation. Since 1996, Bob Sabiston has developed Rotoshop, a set of tools for drawing and positioning graphics on top of video frames. He refined the software to make the ambitious animated feature *Waking Life* (p. 383). Ed Burton's *MOOVL* software extends ideas from the *Sodaconstructor* (p. 263) to a drawing program in which visual elements are aware of their relation to their environment and other elements. In *MOOVL*, shapes can be drawn, connected, and trained to move. The behavior can be mediated via changes in the gravity and other aspects of the simulation. The *Mobility Agents* software created by John F. Simon, Jr. (1989–2005) augments lines drawn by hand with additional lines drawn by the software. Drawn lines are augmented by or replaced with lines that correspond to the angle and speed at which the initial lines are drawn. Zach Lieberman's *Drawn* software (2005) explores a hybrid space of physical materials and software animation. Marks made on paper with a brush and ink are brought to life through the clever use of a video camera and computer vision techniques. The camera takes an image and the software calculates a mark's location and shape, at which point the mark can respond like any other reactive software form.

Artists explore software as a medium for pushing drawing in new directions. Drawing with software provides the ability to integrate time, response, and behavior with drawn marks. Information from the mouse (introduced in Input 5, p. 245) can be combined with techniques of motion (introduced in Motion 2, p. 291) to produce animated drawings that capture the kinetic gestures of the hand and reinterpret them as intricate motion. Other unique inputs, such as voice captured through a microphone and body gestures captured through a camera, can be used to control drawings.

Active tools

Software drawing instruments can change their form in response to gestures made by the hand. Comparison of `mouseX` and `mouseY` variables with previous mouse values can determine the direction and speed of motion. In the following example, the change in the mouse position between the last frame and current frame sets the size of the ellipse drawn to the screen. If the ellipse does not move, the size reverts to a single pixel.

44-01

```
void setup() {
  size(100, 100);
  smooth();
}

void draw() {
  float s = dist(mouseX, mouseY, pmouseX, pmouseY) + 1;
  noStroke();
  fill(0, 102);
  ellipse(mouseX, mouseY, s, s);
  stroke(255);
  point(mouseX, mouseY);
}
```

Software drawing instruments can follow a rhythm or abide by rules independent of drawn gestures. This is a form of collaborative drawing in which the draftsperson controls some aspects of the image and the software controls others. In the examples that follow, the drawing elements obey their own rules, but the draftsperson controls each element's origin. In the next example, the drawing tool pulses from a small to a large size, supplementing the motion of the hand.

44-02

```
int angle = 0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  fill(0, 102);
}

void draw() {
  // Draw only when mouse is pressed
  if (mousePressed == true) {
    angle += 10;
    float val = cos(radians(angle)) * 6.0;
    for (int a = 0; a < 360; a += 75) {
      float xoff = cos(radians(a)) * val;
      float yoff = sin(radians(a)) * val;
      fill(0);
      ellipse(mouseX + xoff, mouseY + yoff, val/2, val/2);
    }
    fill(255);
    ellipse(mouseX, mouseY, 2, 2);
  }
}
```

In the next example, the `Blade` class defines a drawing tool that creates a growing diagonal line when the mouse is not moving and resets the line to a new position when the mouse moves.

44-03

```
Blade diagonal;

void setup() {
  size(100, 100);
  diagonal = new Blade(30, 80);
}

void draw() {
  diagonal.grow();
}

void mouseMoved() {
  diagonal.seed(mouseX, mouseY);
}
```

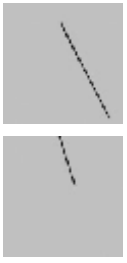
```
class Blade {
  float x, y;

  Blade(int xpos, int ypos) {
    x = xpos;
    y = ypos;
  }

  void seed(int xpos, int ypos) {
    x = xpos;
    y = ypos;
  }

  void grow() {
    x += 0.5;
    y -= 1.0;
    point(x, y);
  }
}
```

44-03 cont.

Active drawings

Individual drawing elements with their own behavior can produce drawings with or without input from a person. These active drawings are a bit like what would result from a raccoon stumbling into a paint tray and then running across pavement. Though created by a series of predetermined rules and actions, the drawings are partially or totally autonomous.

The code for the next example is presented in steps because it's longer than most in the book. Before writing the longer program, we first wrote a small program to test the desired effect. This code displays a line that changes position very slightly with each frame. Over a long period of time, the line's position changes significantly. This is similar to code 32-05 (p. 296), but is more subtle.



```
float x1, y1, x2, y2;

void setup() {
  size(100, 100);
  smooth();
  x1 = width / 4.0;
  y1 = x1;
  x2 = width - x1;
  y2 = x2;
}
```

44-04

```
void draw() {
  background(204);
  x1 += random(-0.5, 0.5);
  y1 += random(-0.5, 0.5);
  x2 += random(-0.5, 0.5);
  y2 += random(-0.5, 0.5);
  line(x1, y1, x2, y2);
}
```

44-04 cont.

If several such lines are drawn, the drawing will degrade over time as each line continues to wander from its original position. In the next example, the code from above was modified to create the `MovingLine` class. Five hundred of these `MovingLine` objects populate the display window. When the lines are first drawn, they vibrate but maintain their form. Over time, the image degrades into chaos as each line wanders across the surface of the window.

```
int numLines = 500;
MovingLine[] lines = new MovingLine[numLines];
int currentLine = 0;

void setup() {
  size(100, 100);
  smooth();
  frameRate(30);
  for (int i = 0; i < numLines; i++) {
    lines[i] = new MovingLine();
  }
}

void draw() {
  background(204);
  for (int i = 0; i < currentLine; i++) {
    lines[i].display();
  }
}

void mouseDragged() {
  lines[currentLine].setPosition(mouseX, mouseY,
                                 pmouseX, pmouseY);
  if (currentLine < numLines - 1) {
    currentLine++;
  }
}
```

44-05

44-05
cont.

```
class MovingLine {
  float x1, y1, x2, y2;

  void setPosition(int x, int y, int px, int py) {
    x1 = x;
    y1 = y;
    x2 = px;
    y2 = py;
  }

  void display() {
    x1 += random(-0.1, 0.1);
    y1 += random(-0.1, 0.1);
    x2 += random(-0.1, 0.1);
    y2 += random(-0.1, 0.1);
    line(x1, y1, x2, y2);
  }
}
```

The next example shows a simple animation tool that displays a continuous cycle of twelve images. Each image is displayed for 100 milliseconds (one tenth of a second) to create animation. While each image is displayed, it's possible to draw directly into it by pressing the mouse and moving the cursor.

44-06

```
int currentFrame = 0;
PImage[] frames = new PImage[12];
int lastTime = 0;

void setup() {
  size(100, 100);
  strokeWeight(4);
  smooth();
  background(204);
  for (int i = 0; i < frames.length; i++) {
    frames[i] = get();  // Create a blank frame
  }
}

void draw() {
  int currentTime = millis();
  if (currentTime > lastTime+100) {
    nextFrame();
    lastTime = currentTime;
  }
```

44-06 cont.

```
  if (mousePressed == true) {
    line(pmouseX, pmouseY, mouseX, mouseY);
  }
}

void nextFrame() {
  frames[currentFrame] = get();  // Get the display window
  currentFrame++;  // Increment to next frame
  if (currentFrame >= frames.length) {
    currentFrame = 0;
  }
  image(frames[currentFrame], 0, 0);
}
```

Exercises

1. *Design and program your own active drawing instrument.*
2. *Design and program visual elements that change after they have been drawn to the display window.*
3. *Extend code 44-06 into a more complete animation program.*

Notes

1. Norman McLaren, "Animated Films," in *Experimental Animation*, edited by Robert Russett and Cecile Starr (Da Capo Press, 1976), p. 122.

Output 2: File Export

This unit introduces the formatting of data and the writing of files.

Syntax introduced:

```
nf(), saveStrings(),
PrintWriter, createWriter(),
PrintWriter.flush(), PrintWriter.close(), exit()
```

Digital files on computers are not tangible like their paper namesakes, and they don't sit in file cabinets for years collecting dust. A digital file is a sequence of bytes at a location on the computer's disk. Despite the diverse content stored in digital files, the material of each is the same—a sequence of 1s and 0s. Almost every task performed with computers involves working with files. For example, before a text document is written, the text editor application file must be read and a new data file created to store the content. When the information is saved, the file is given a name and written to disk for later retrieval.

The primary reason to save a file is to store data so that it's available after a program stops running. When running, a program uses part of the computer's memory to store its data temporarily. When the program is stopped, the program gives control of this memory back to the operating system so other programs can access it. If the data created by the program is not saved to a file, it is lost when the program closes.

All software files have a format, a convention for ordering data so that software applications know how to interpret the data when it is read from memory. Some common formats include TXT for plain text files, MP3 for storing sound, and EXE for executable programs on Windows. Common formats for image data are JPEG and GIF (pp. 95, 96) and common formats for text documents are DOC and RTF. The XML format has become popular in recent years as a general-purpose data format that can be extended to hold specific types of data in an easy-to-read file.

Formatting data

Text files often contain characters that are not visible (referred to as nonprintable) and are used to define the spacing of the visible characters. The two most common are *tab* and *new line*. These characters can be represented in code as `\t` and `\n`, respectively. The combination of the `\` (backslash) character with another is called an escape sequence. These escape sequences are treated as one character by the computer. The backslash begins the escape sequence and the second character defines the meaning. It's often useful to put escape sequences in your programs to make the files easier to read or to make it easier to load them back into a program and separate the data elements.

```
// Prints "tab     space"                                    45-01
println("tab\tspace");

// Prints each word after "\n" on a new line:
// line1
// line2
// line3
println("line1\nline2\nline3");
```

Data can also be formatted with functions such as `nf()`. There are two versions of this function:

```
nf(intValue, digits)
nf(floatValue, left, right)
```

The *`intValue`* parameter is an integer number to be formatted, and the *`digits`* parameter is the total number of digits in the formatted number. The *`floatValue`* parameter is a floating-point number to be formatted; the *`left`* parameter sets the number of digits to the left of the decimal, and the *`right`* parameter sets the number of digits to the right of the decimal. Setting either digits parameter to zero means "any" number of digits. In addition to formatting `int` and `float` data, the `nf()` function converts the data into the `String` type so it can be output to the console or saved to a text file.

```
println(nf(200, 10));        // Prints "0000000200"          45-02
println(nf(40, 5));          // Prints "00040"
println(nf(90, 3));          // Prints "090"
println(nf(200.94, 10, 4));  // Prints "0000000200.9400"
println(nf(40.2, 5, 3));     // Prints "00040.200"
println(nf(9.012, 0, 5));    // Prints "9.01200"
```

Exporting files

Saving files is a useful way to store data so it can be viewed after a program has stopped running. Data can either be saved continuously while the program runs or stored in variables while the program is running; and then it can be saved to a file in one batch.

The `saveStrings()` function writes an array of strings to a file, with each string written to a new line. This file is saved to the sketch's folder and can be accessed by selecting the "Show Sketch Folder" item from the Sketch menu. The following example uses the `saveStrings()` function to write data created while drawing lines to the screen. Each time a mouse button is pressed, a new value is added to the `x[]` and `y[]` arrays, and when a key is pressed the data stored in these arrays is written to a file called *lines.txt*. The `exit()` function then stops the program.

```
int[] x = new int[0];
int[] y = new int[0];

void setup() {
  size(100, 100);
}

void draw() {
  background(204);
  stroke(0);
  noFill();
  beginShape();
  for (int i = 0; i < x.length; i++) {
    vertex(x[i], y[i]);
  }
  endShape();
  // Show the next segment to be added
  if (x.length >= 1) {
    stroke(255);
    line(mouseX, mouseY, x[x.length-1], y[x.length-1]);
  }
}

void mousePressed() {  // Click to add a line segment
  x = append(x, mouseX);
  y = append(y, mouseY);
}

void keyPressed() {  // Press a key to save the data
  String[] lines = new String[x.length];
  for (int i = 0; i < x.length; i++) {
    lines[i] = x[i] + "\t" + y[i];
  }
  saveStrings("lines.txt", lines);
  exit();  // Stop the program
}
```

The `PrintWriter` class provides another way to export files. Instead of writing the entire file at one time as `saveStrings()` does, the `createWriter()` method opens a file to write to and allows data to be added continuously to the file while the program is running. To make the file save correctly, it's necessary to use the `flush()` method to write any remaining data to the file. The `close()` method is also needed to finish writing the file properly. The following example uses the `PrintWriter` to save the cursor position to a file while a mouse button is pressed.

```
PrintWriter output;                                                    45-04

void setup() {
  size(100, 100);
  // Create a new file in the sketch directory
  output = createWriter("positions.txt");
  frameRate(12);
}

void draw() {
  if (mousePressed) {
    point(mouseX, mouseY);
    // Write the coordinate to a file with a
    // "\t" (TAB character) between each entry
    output.println(mouseX + "\t" + mouseY);
  }
}

void keyPressed() {   // Press a key to save the data
  output.flush();     // Write the remaining data
  output.close();     // Finish the file
  exit();             // Stop the program
}
```

The file created with the previous program has a simple format. The x-coordinate of the cursor is written followed by a tab, then followed by the y-coordinate. Code 46-01 (p. 429) shows how to load this file back into another sketch and use the data to redraw the saved points.

The next example is a variation of the previous one, but uses the spacebar and Enter key to control when data is written to the file and when the file is closed. When a key is pressed, the character is added to the `letters` variable. When the spacebar is pressed, the `String` is written to the *words.txt* file. When the Enter key is pressed, the file is flushed, then is closed, and the program exits.

```
PFont font;                                                            45-05
String letters = "";
PrintWriter output;

void setup() {
  size(100, 100);
  fill(0);
  font = loadFont("Eureka-24.vlw");
  textFont(font);
```

45-05 cont.

```
  // Create a new file in the sketch directory
  output = createWriter("words.txt");
}

void draw() {
  background(204);
  text(letters, 5, 50);
}

void keyPressed() {
  if (key == ' ') {           // Spacebar pressed
    output.println(letters);  // Write data to words.txt
    letters = "";             // Clear the letter String
  } else {
    letters = letters + key;
  }
  if (key == ENTER) {
    output.flush();           // Write the remaining data
    output.close();           // Finish the file
    exit();                   // Stop the program
  }
}
```

Exercises

1. *Use `nf()` to reformat the value 12.2 into these configurations: 0012.20000, 12.20, 00012.2.*
2. *While a program is running, save every letter key and the time it was pressed into a file named* timekeys.txt.
3. *Using code 45-03 as a base, make a Bézier curve editor that exports its geometry to a file.*

```
39061 45258 +39.166759 -084.538220 P CINCINNATI
39061 45262 +39.166759 -084.538220 P CINCINNATI
39061 45263 +39.166759 -084.538220 U CINCINNATI
39061 45264 +39.166759 -084.538220 U CINCINNATI
39061 45267 +39.166759 -084.538220 U CINCINNATI
39061 45268 +39.166759 -084.538220 U CINCINNATI
39061 45269 +39.166759 -084.538220 U CINCINNATI
39061 45270 +39.166759 -084.538220 U CINCINNATI
39061 45271 +39.166759 -084.538220 U CINCINNATI
39061 45273 +39.166759 -084.538220 U CINCINNATI
39061 45274 +39.166759 -084.538220 U CINCINNATI
39061 45275 +38.946921 -083.862877 P CINCINNATI
39015 45277 +39.166759 -084.538220 U CINCINNATI
39061 45296 +39.166759 -084.538220 U CINCINNATI
39061 45298 +39.166759 -084.538220 U CINCINNATI
39061 45299 +39.262158 -084.509268 U CINCINNATI
39061 45301 +39.706459 -084.016233 P ALPHA
39057 45302 +40.407239 -084.203271   ANNA
39149 45303 +40.214675 -084.653188 A NSONIA
39037 45304 +40.126915 -084.539928   ARCANUM
39037 45305 +39.632829 -084.049985   BELLBROOK
39057 45306 +40.439778 -084.189245   BOTKINS
39149 45307 +39.575597 -083.715323 P BOWERSVILLE
39057 45308 +40.115737 -084.279352   BRADFORD
39109 45309 +39.836157 -084.330392   BROOKVILLE
39113 45310 +40.354106 -084.643532 P BURKETTSVILLE
39107 45311 +39.640948 -084.647001   CAMDEN
39135 45312 +40.066567 -084.081610   CASSTOWN
39109 45314 +39.747459 -083.759973   CEDARVILLE
39057 45315 +39.854454 -084.340345   CLAYTON
39113 45316 +39.795971 -083.819766 P CLIFTON
39057 45317 +40.137029 -084.046873   CONOVER
39109 45318 +40.124386 -084.281167   COVINGTON
39109 45319 +39.918936 -083.944909 P DONNELSVILLE
39023 45320 +39.774850 -084.674889   EATON
39135 45321 +39.872177 -084.681067   ELDORADO
39135 45322 +39.877005 -084.331945   ENGLEWOOD
39113 45323 +39.857967 -083.933431   ENON
39023 45324 +39.728549 -084.014834   FAIRBORN
39057 45325 +39.781301 -084.413970   FARMERSVILLE
39113 45326 +40.144491 -084.100988   FLETCHER
39109 45327 +39.747497 -084.396912   GERMANTOWN
39113 45328 +40.114729 -084.493439 P GETTYSBURG
39037 45329 +40.135426 -084.619129   GORDON
39037 45330 +39.641558 -084.527613 P GRATIS
39135 45331 +40.156478 -084.649068   GREENVILLE
39037 45332 +39.993964 -084.783714   HOLLANSBURG
39037 45333 +40.248489 -084.345612   HOUSTON
39149 45334 +40.434921 -084.058495   JACKSON CENTER
39149 45335 +39.674084 -083.766709   JAMESTOWN
39057 45336 +40.441325 -084.262602 P KETTLERSVILLE
39149 45337 +39.985177 -084.399364   LAURA
39109 45338 +39.780916 -084.567331   LEWISBURG
39135 45339 +40.005764 -084.351781   LUDLOW FALLS
39109 45340 +40.364305 -084.056464   MAPLEWOOD
39149 45341 +39.878918 -084.021827   MEDWAY
39023 45342 +39.641658 -084.274640   MIAMISBURG
39113 45343 +39.750471 -084.268593 P MIAMISBURG
39113 45344 +39.959249 -083.986855   NEW CARLISLE
39023 45345 +39.800056 -084.327000   NEW LEBANON
39113 45346 +39.982103 -084.705736   NEW MADISON
39037 45347 +39.783378 -084.668892   NEW PARIS
39135 45348 +40.316833 -084.633911   NEW WESTON
39037 45349 +39.989309 -083.938933 P NORTH HAMPTON
39023 45350 +40.135426 -084.619129 P NORTH STAR
39037 45351 +40.340616 -084.496342 P OSGOOD
39037 45352 +40.050323 -084.745706 P PALESTINE
39037 45353 +40.295220 -084.032272 P PEMBERTON
39149 45354 +39.905385 -084.402785 P PHILLIPSBURG
39113 45356 +40.123618 -084.228811   PIQUA
39109 45358 +39.987043 -084.486582 P PITSBURG
39037 45359 +40.050483 -084.348699   PLEASANT HILL
39109 45360 +40.330713 -084.092589 P PORT JEFFERSON
39149 45361 +39.963492 -084.414484 P POTSDAM
39109 45362 +40.287621 -084.637078   ROSSBURG
39037 45363 +40.263253 -084.263227   RUSSIA
39149 45365 +40.293558 -084.209198   SIDNEY
39149 45367 +40.333611 -084.218308 U SIDNEY
39149 45368 +39.854349 -083.665280   SOUTH CHARLESTON
39023 45369 +39.957723 -083.614481   SOUTH VIENNA
39023 45370 +39.608285 -084.025972   SPRING VALLEY
39057 45371 +39.941967 -084.166260   TIPP CITY
39109 45372 +40.013871 -083.833250 P TREMONT CITY
39023 45373 +40.062621 -084.226398   TROY
39109 45374 +40.039970 -084.229799 U TROY
39109 45377 +39.889006 -084.242243   VANDALIA
39113 45378 +39.897035 -084.499044 P VERONA
39135 45380 +40.253040 -084.523891   VERSAILLES
39037 45381 +39.750097 -084.537597   WEST ALEXANDRIA
39135 45382 +39.881330 -084.621617   WEST MANCHESTER
39135 45383 +39.987910 -084.350107   WEST MILTON
39109 45384 +39.712811 -083.878088 P WILBERFORCE
39057 45385 +39.684731 -083.908130   XENIA
39057 45387 +39.760531 -083.883600   YELLOW SPRINGS
39057 45388 +40.321853 -084.484466   YORKSHIRE
39037 45389 +40.056400 -084.025444 P CHRISTIANSBURG
39021 45390 +40.211787 -084.758818   UNION CITY
39037 45401 +39.750471 -084.268593 P DAYTON
39113 45402 +39.756658 -084.181639   DAYTON
39113 45403 +39.764658 -084.150738   DAYTON
39113 45404 +39.794958 -084.163589   DAYTON
39113 45405 +39.789857 -084.217391   DAYTON
39113 45406 +39.782457 -084.239391   DAYTON
39113 45407 +39.758658 -084.226041   DAYTON
```

Input 6: File Import

This unit focuses on loading files and accessing the file data.

Syntax introduced:
`loadStrings(), split(), splitTokens(), WHITESPACE`

Output 2 (p. 421) explained how to export files, and this unit complements it by demonstrating how to load files. Files are the easiest way to store and load data, but before you load a data file into a program, it's essential to know how the file is formatted. In a plain text file, the control characters for tab and new line (p. 421) are used to differentiate and align the data elements. Separating the individual elements with a tab or space character and each line with a new line character is a common formatting technique. Here's one example excerpted from a data file:[1]

```
00214 +43.005895 -071.013202 U PORTSMOUTH 33 015
00215 +43.005895 -071.013202 U PORTSMOUTH 33 015
00501 +40.922326 -072.637078 U HOLTSVILLE 36 103
00544 +40.922326 -072.637078 U HOLTSVILLE 36 103
00601 +18.165273 -066.722583   ADJUNTAS   72 001
00602 +18.393103 -067.180953   AGUADA     72 003
00603 +18.455913 -067.145780   AGUADILLA  72 005
```

If you see a file formatted in a similar way, you can use a text editor to tell whether there are tabs or spaces between the elements by moving the cursor to the beginning of a line and using the arrow keys to navigate left or right through the characters. If the cursor jumps from one element to another, there is a tab between the elements; if the cursor moves via a series of steps through the whitespace, spaces were used to format the data. In addition to knowing how the data elements are separated, it's essential to know how many data elements each line contains and the data type of each element. For example, the file above has data that should be stored as `String`, `int`, and `float` variables.

In addition to loading data from plain text files, it's common to load data from XML files. XML is a file structure that is based on "tagging" information, similar to its cousin HTML. It defines a structure for ordering data, but leaves the content and categories of the data elements open. For example, in an XML structure designed for storing book information, each element might have an entry for title and publisher:

```
<book>
    <title>Processing</title>
    <publisher>MIT Press</publisher>
</book>
```

In an XML structure designed for storing a list of websites, each element might have an entry for the name of the website and the URL.

```
<website>
    <name>Processing.org</name>
    <url>http://processing.org</url>
</website>
```

In these two examples, notice that the names of the element tags are different, but the structure is the same. Each entry is defined with a tag to begin the data and a corresponding tag to end the entry. Because the tag for each data element describes the type of content, XML files are often more self-explanatory than files delimited by tabs. The XML library included with Processing can load and parse simple, strictly-formatted XML files. Contributed libraries have been developed with a broader set of features.

Tab-delimited and XML data are useful in different contexts. Many "feeds" available from the Web are available in XML format. These include weather service updates from the NOAA and the RSS feeds common to many websites. In these cases, the data is both varied and hierarchical, making it suitable for XML. For information exported from a database, a tab-delimited file is more appropriate, because the additional metadata included in XML wastes considerable space and takes longer to load into a program. For example, the excerpt presented at the beginning of this unit is from a file that contains 40,000 lines. Because the data comprises seven straightforward columns, adding additional tags to make this XML would make it unnecessarily burdensome and slow.

Loading numbers

The easiest way to bring external data into Processing is to save it as a file in TXT format. The file can then be loaded and parsed to extract the individual data elements. A TXT file format stores only plain text characters, which means there is no formatting such as bold, italics, and colors.

Numbers are stored in files as characters. The easiest way to load them into Processing is to treat the numbers temporarily as a string before converting them to floating-point or integer variables. A file containing numbers can be loaded into Processing with the `loadStrings()` function. This function reads the contents of a file and creates a string array of its individual lines—one array element for each line of the file. As with any media loaded into Processing, the file must be located in the sketch's *data* folder. For example, if the text file *numbers.txt* is in the current sketch's data folder, its data can be read into Processing with this line of code:

```
String[] lines = loadStrings("numbers.txt");
```

The `lines[]` array is first declared and then assigned the `String` array created by the `loadStrings()` function. It holds the contents of the file, with each element in the

array containing one line of the text in the file. This code reads through each element of the array and prints its contents to the Processing console:

```
for (int i = 0; i < lines.length; i++) {
  println(lines[i]);
}
```

The following example loads the text file created with code 45-04 (p. 424). This file contains the `mouseX` and `mouseY` variable separated by a tab and formatted like this:

```
x1	y1
x2	y2
x3	y3
x4	y4
x5	y5
```

This program is designed to read the entire file into an array; then it reads each line of the array and extracts the two coordinate values into another array. The file checks to make sure the data is formatted as expected by confirming that there are two elements on each line, then converts these elements to integer values and uses them to draw a point to the screen.

46-01

```
String[] lines = loadStrings("positions.txt");

for (int i = 0; i < lines.length; i++) {
  // Split this line into pieces at each tab character
  String[] pieces = split(lines[i], '\t');
  // Take action only if there are two values on the line
  // (this will avoid blank or incomplete lines)
  if (pieces.length == 2) {
    int x = int(pieces[0]);
    int y = int(pieces[1]);
    point(x, y);
  }
}
```

The `split()` function is used to divide each line of the text file into its separate elements. This function splits a string into pieces using a character or string as the divider.

```
split(str, delim)
```

The *`str`* parameter must be a `String`, but the *`delim`* parameter can be a `char` or `String` and does not appear in the returned `String[]` array.

```
String s = "a, b";                                                    46-02
String[] p = split(s, ", ");
println(p[0]);  // Prints "a"
println(p[1]);  // Prints "b"
```

The `splitTokens()` function allows you to split a `String` at one or many character "tokens." There are two versions of this function:

```
splitTokens(str)
splitTokens(str, tokens)
```

The *`tokens`* parameter is a `String` containing a list of characters that are used to separate the line. If the *`tokens`* parameter is not used, all whitespace characters (space, tab, new line, etc.) are used as delimiters.

```
String t = "a b";                                                     46-03
String[] q = splitTokens(t);
println(q[0]);  // Prints "a"
println(q[1]);  // Prints "b"
```

The following example demonstrates the flexibility of `splitTokens()`. When ", " is used as the *`tokens`* parameter, it doesn't matter in what order the comma and space appear in the file, or whether there is just a comma or just a space.

```
String s = "a, b c   ,,d  ";        // Despite the bad formatting,    46-04
String[] p = splitTokens(s, ", ");  // the data is parsed correctly
println(p[0]);  // Prints "a"
println(p[1]);  // Prints "b"
println(p[2]);  // Prints "c"
println(p[3]);  // Prints "d"
```

The same data file used in code 46-01 can be used to display the points from the file in the sequence in which they were originally drawn. Adding `setup()` and `draw()` requires the `lines[]` array to be declared at the beginning of the sketch. Rather than every point being drawn inside a `for` structure, only one point is drawn each time the `draw()` is run.

```
String[] lines;                                                       46-05
int index = 0;

void setup() {
  lines = loadStrings("positions.txt");
  frameRate(12);
}
```

```
void draw() {                                                    46-05
  if (index < lines.length) {                                    cont.
    String[] pieces = split(lines[index], '\t');
    if (pieces.length == 2) {
      int x = int(pieces[0]);
      int y = int(pieces[1]);
      point(x, y);
    }
    // Go to the next line for the next run through draw()
    index = index + 1;
  }
}
```

The code for reading other data formats will be very similar to the examples above.

Loading characters

Loading numbers from a file is similar to loading text data. Files usually contain multiple kinds of data, so it's important to know what kind is inside a file so that it can be parsed into variables of the appropriate type (p. 37).

The following example loads data about cars. In the file used for this example, text data is mixed with integer and floating-point numbers:

```
ford galaxie 500      15    8    429    198    4341    10     70    1
chevrolet impala      14    8    454    220    4354    9      70    1
plymouth fury iii     14    8    440    215    4312    8.5    70    1
pontiac catalina      14    8    455    225    4425    10     70    1
```

This small excerpt of a file[2] shows its content and formatting. Each element is separated with a tab and corresponds to a different aspect of each car. This file stores the miles per gallon, cylinders, displacement, etc., for more than 400 different cars. A `Record` class was created to load this data and store the information for each entry. An array of `Record` objects was created for all 400 cars. The first `for` loop loads the data into an array of objects, and the second `for` loop lists the data the console.

```
Record[] records;                                                46-06
int recordCount;

void setup() {
  String[] lines = loadStrings("cars2.tsv");
  records = new Record[lines.length];
  for (int i = 0; i < lines.length; i++) {
    String[] pieces = split(lines[i], '\t');  // Load data into array
```

```
    if (pieces.length == 9) {
      records[recordCount] = new Record(pieces);
      recordCount++;
    }
  }
  for (int i = 0; i < recordCount; i++) {
    println(i + " -> " + records[i].name);  // Print name to console
  }
}

class Record {
  String name;
  float mpg;
  int cylinders;
  float displacement;
  float horsepower;
  float weight;
  float acceleration;
  int year;
  float origin;

  public Record(String[] pieces) {
    name = pieces[0];
    mpg = float(pieces[1]);
    cylinders = int(pieces[2]);
    displacement = float(pieces[3]);
    horsepower = float(pieces[4]);
    weight = float(pieces[5]);
    acceleration = float(pieces[6]);
    year = int(pieces[7]);
    origin = float(pieces[8]);
  }
}
```

46-06 cont.

This example only shows how to load the data into the program. The `Record` class could be extended to include an image or vertex model of each car, which would enable the creation of a visual database that could be navigated using the statistics and design of each vehicle.

The next example loads the text of a book into a program and counts the number of words, printing words longer than ten letters to the console. It uses a built-in variable called `WHITESPACE`, a string that contains the most common control characters that create whitespace within a text file. It is literally the string "` \t\n\r\f\u00A0`", which includes the common escape sequences for tab, new line, carriage return, formfeed, and the Unicode "nonbreaking space" character (Appendix C, p. 664). The `WHITESPACE`

constant differentiates between the individual elements of the book's text. The book loaded into the program comes from the Gutenberg archive,[3] which formats its documents so the actual text of the book begins with `*** START` and ends with `*** END`. These specific character sequences are used within the program to set when it starts and stops counting words.

46-07

```
String[] lines = loadStrings("2895.txt");
int totalCount = 0;  // Total word count for entire book
boolean started = false;  // Ignore lines until the *** START line

for (int i = 0; i < lines.length; i++) {
  if (lines[i].startsWith("*** START")) {  // Start parsing text
    started = true;
  } else if (lines[i].startsWith("*** END")) {  // Stop parsing text
    started = false;
  } else if (started == true) {  // If we're in the useful region
    // List of characters and punctuation to ignore between
    // letters. WHITESPACE is all the whitespace characters
    String separators = WHITESPACE + ",;.:?()\"-";
    // Split the line anywhere that we see one or more of
    // these separators
    String[] words = splitTokens(lines[i], separators);
    // Add this number to the total
    totalCount += words.length;
    // Go through the list of words on the line
    for (int j = 0; j < words.length; j++) {
      String word = words[j].toLowerCase();
      if (word.length() > 10) {
        println(word);  // Print word if longer than ten letters
      }
    }
  }
}

// How many words are in the entire book?
println("This book has " + totalCount + " total words.");
```

When this program is run, the last fifteen lines printed to the console are:

```
requirements
considerable
requirements
confirmation
contributions
```

```
solicitation
requirements
prohibition
unsolicited
international
information
distributed
necessarily
information
This book has 194700 total words.
```

Exercises

1. *Write a program to load and display the data saved in code 45-03 (p. 423).*
2. *Write a program to load and display the data saved in code 45-05 (p. 424).*
3. *Select a data set from* http://lib.stat.cmu.edu/datasets *and write a program to load and display the data.*

Notes

1. http://www.census.gov/geo/www/tiger/zip1999.html.
2. From the StatLib Datasets Archive at Carnegie Mellon University, http://lib.stat.cmu.edu/datasets/cars.data. More information about this dataset can be found at http://lib.stat.cmu.edu/datasets/cars.desc.
3. http://www.gutenberg.org/files/2895/2895.txt.

Input 7: Interface

This unit introduces and discusses code for graphical interface elements.

The graphical user interface (GUI), also known as the direct manipulation interface, helped bring computers out of laboratories and into homes, offices, and schools. The combination of the mouse and graphical interfaces has made computer use intuitive. Common navigation techniques such as pointing, clicking, and dragging all require a device like the mouse that controls an on-screen cursor. Most GUIs are comprised of icons representing the hierarchy of files and folders on the hard drive. The user performs actions by selecting icons and moving them directly with the cursor.

Before pointing devices were developed, the most common way to interface with a computer was through a command line interface (CLI). A CLI requires text commands such as `cp` (copy), `mv` (move), and `mkdir` (make directory) to perform actions on files. Moving the file *data.txt* from its current folder to a different folder named *base* is achieved in UNIX, known for its CLI, with the text:

```
mv data.txt base/
```

Unlike the GUI, in which the *data.txt* icon is dragged to a folder icon titled *base*, working professionally with a CLI requires the user to maintain a mental model of the folder structures and remember the names of commands. Some actions are easier to perform with one style of interface, some with the other; both have their benefits and difficulties.

Operating systems like Mac OS and Windows have a distinct visual appearance and style of interaction, which is determined by the sum of the behaviors of individual elements in the interface. The visual difference between operating systems is obvious. For example, using Windows NT feels like working inside a concrete bunker, while earlier versions of Mac OS X resembled working inside a brightly lit candy store. The different style of interaction required by each GUI is less obvious but more important. Details such as the way in which a window opens, or how a file is deleted, create the dynamics for the environment we mentally inhabit while using a computer.

The GUI has evolved continuously over the last thirty years, but the basic metaphor remains unchanged. This standard interface method is referred to as WIMP (an acronym for Windows, Icons, Mouse, and Pointer). There have been fascinating explorations into alternative computer interfaces including Zooming User Interfaces (ZUIs) such as the Pad interface model and its derivatives. In contrast to Windows interfaces where elements open and close, a zooming interface allows the user to zoom out to get an overview of the computer's contents and zoom in to view individual data elements. This technique provides a map to the complete data landscape that other windowing systems obscure. The Lifestreams project is another alternative interface. It was developed as a networked replacement for the software desktop with the goal of reducing the time spent managing documents while simultaneously making them more accessible. A lifestream

is an ordered stream of digital information including pictures, movies, Emails, and bills. The files in an individual's stream are structured in the order of creation, starting with their first document and continuing through their entire life up to the present. As these and other exploratory GUI projects are emerging from the research community, the video game industry is continuously experimenting with interface techniques used to navigate the ever more complex information contained in games.

Writing GUI programs can be more difficult than writing CLI programs because of the additional code needed to draw elements to the screen and define their behavior. Specialized libraries for creating GUI elements help reduce the time spent coding. Microsoft, for example, developed the Visual Basic programming environment to assist people in assembling windows with menus, buttons, and behaviors; it allows them to select from available graphic elements and assign them behaviors with menus. The Adobe Flash software has a `Button` object that simplifies the creation of interface buttons. Creating a program with a basic interface requires understanding interface techniques and common GUI elements such as buttons, check boxes, radio buttons, and scrollbars. With clear knowledge of how each GUI element works, the programmer can understand how to modify them to improve the way people interface with computers.

Rollover, Button, Dragging

The first step in building an interface element is to make the shape aware of the mouse. The two shapes that will most easily recognize the mouse within their boundaries are the circle and the rectangle. The `OverCircle` and `OverRect` classes presented below have been written to make it clear when the mouse is over these elements.

The `OverCircle` class has four fields. They set the x-coordinate, y-coordinate, diameter, and gray value. The `update()` method is run when the mouse is over the element, and the `display()` method draws the element to the screen. The position and size of the circle are set within the constructor, and the default gray value is set to black. The `dist()` function within `update()` calculates the distance from the mouse to the center of the circle; if the distance is less than the circle's radius, the `gray` value is set to white.

47-01

```
class OverCircle {
  int x, y;      // The x- and y-coordinates
  int diameter;  // Diameter of the circle
  int gray;      // Gray value

  OverCircle(int xp, int yp, int d) {
    x = xp;
    y = yp;
    diameter = d;
    gray = 0;
  }
```

```
  void update(int mx, int my) {
    if (dist(mx, my, x, y) < diameter/2) {
      gray = 255;
    } else {
      gray = 0;
    }
  }

  void display() {
    fill(gray);
    ellipse(x, y, diameter, diameter);
  }
}
```

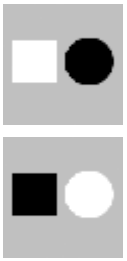
47-01 cont.

The fields and methods for the `OverRect` class are identical to those in `OverCircle`, but the `size` field now sets the width and height of the rectangle rather than the diameter of a circle. The relational expression inside `update()` tests to see if the incoming `mouseX` and `mouseY` values are within the rectangle.

```
class OverRect {
  int x, y;  // The x- and y-coordinates
  int size;  // Dimension (width and height) of the rectangle
  int gray;  // Gray value

  OverRect(int xp, int yp, int s) {
    x = xp;
    y = yp;
    size = s;
    gray = 0;
  }

  void update(int mx, int my) {
    if ((mx > x) && (mx < x+size) && (my > y) && (my < y+size)) {
      gray = 255;
    } else {
      gray = 0;
    }
  }

  void display() {
    fill(gray);
    rect(x, y, size, size);
  }
}
```

47-02

In the next example, objects are created from the `OverRect` and `OverCircle` class and are placed within the display window. When the mouse moves over each element, the color changes from black to white.



47-03

```
// Requires the OverRect and OverCircle classes

OverRect r = new OverRect(9, 30, 36);
OverCircle c = new OverCircle(72, 48, 40);

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  background(204);
  r.update(mouseX, mouseY);
  r.display();
  c.update(mouseX, mouseY);
  c.display();
}
```

Both of these classes can be extended further. For example, changing the `update()` method for both `OverCircle` and `OverRect` can create a smooth value transition when the mouse is over the shape. The following code fragment shows how to do this for `OverCircle`. Use the `update()` method below in place of the original `update()` in the last example to see the difference. This gradual transition detail gives interface elements a subtlety that enhances their behavior.

47-04

```
void update(int mx, int my) {
  if (dist(mx, my, x, y) < diameter/2) {
    if (gray < 250) {
      gray++;
    }
  } else {
    if (gray > 0) {
      gray--;
    }
  }
}
```

The code from `OverRect` can be enhanced to create a rectangular button element. The `Button` class is distinct from the `OverRect` class in that it has an additional state. While

OverRect acknowledges when the mouse is over the shape, the Button class makes it possible to determine whether the mouse is over the shape and when the mouse clicks on the shape. If the mouse is over the button it can trigger an event, and when the mouse is pressed it can trigger a separate event.

Like OverRect, the Button class has a position, a size, and a default gray value. The constructor for Button receives six parameters; the last three set the default gray value, the gray value when the mouse is over the button, and the gray value when the mouse is over the button and is pressed. The update() method sets the boolean field over to true or false depending on the location of the mouse. The press() method should be run from within the main program's mousePressed() function. When run, it sets the boolean field press to true if the mouse is currently over the button and to false if not. The release() method should be run from within the main program's mouseReleased() function. When the mouse is released the boolean field pressed is set to false, which prepares the button to be pressed again. The display() method draws the button to the screen and sets its gray value based on the current status of the mouse in relation to the button.

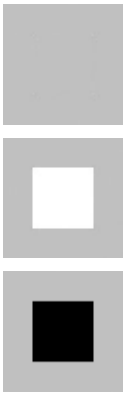
47-05

```
class Button {
  int x, y;                 // The x- and y-coordinates
  int size;                 // Dimension (width and height)
  color baseGray;           // Default gray value
  color overGray;           // Value when mouse is over the button
  color pressGray;          // Value when mouse is over and pressed
  boolean over = false;     // True when the mouse is over
  boolean pressed = false;  // True when the mouse is over and pressed

  Button(int xp, int yp, int s, color b, color o, color p) {
    x = xp;
    y = yp;
    size = s;
    baseGray = b;
    overGray = o;
    pressGray = p;
  }

  // Updates the over field every frame
  void update() {
    if ((mouseX >= x) && (mouseX <= x+size) &&
        (mouseY >= y) && (mouseY <= y+size)) {
      over = true;
    } else {
      over = false;
    }
  }
```

47-05 cont.

```
  boolean press() {
    if (over == true) {
      pressed = true;
      return true;
    } else {
      return false;
    }
  }

  void release() {
    pressed = false;  // Set to false when the mouse is released
  }

  void display() {
    if (pressed == true) {
      fill(pressGray);
    } else if (over == true) {
      fill(overGray);
    } else {
      fill(baseGray);
    }
    stroke(255);
    rect(x, y, size, size);
  }
}
```

To use the `Button` class, add it to a sketch and call its methods from within the mouse event functions. Like most objects, it should be created within `setup()` and updated and displayed within `draw()`. The methods of the object related to the mouse status must be explicitly run from within the `mousePressed()` and `mouseReleased()` functions.



47-06

```
// Requires the Button class

Button button;

void setup() {
  size(100, 100);
  // Inputs: x, y, size,
  // base color, over color, press color
  button = new Button(25, 25, 50,
                      color(204), color(255), color(0));
}
```

47-06 cont.

```
void draw() {
  background(204);
  stroke(255);
  button.update();
  button.display();
}

void mousePressed() {
  button.press();
}

void mouseReleased() {
  button.release();
}
```

Buttons are typically used to trigger events or update values. The previous example showed how to use a `Button` object in isolation, and the next shows how to utilize it to modify the flow of a program. In this example, the buttons turn white when the mouse rolls over them, giving visual feedback that shows they are active elements. When the mouse is pressed over a button, the button turns black to provide feedback. When a button is selected, it updates the `mode` variable in the program. The left button sets `mode` to 1, the middle sets it to 2, and the right button sets it to 3. In the program's `draw()` method, different lines of code are run depending on the value of this variable. In the following example, only the position of an ellipse changes, but the different modes could be used to move between scenes in a complex animation or to change the values of one or more variables.

47-07

```
// Requires the Button class

Button button1, button2, button3;
int mode = 1;

void setup() {
  size(100, 100);
  smooth();
  color gray = color(204);
  color white = color(255);
  color black = color(0);
  button1 = new Button(10, 80, 10, gray, white, black);
  button2 = new Button(25, 80, 10, gray, white, black);
  button3 = new Button(40, 80, 10, gray, white, black);
}
```

47-07
cont.

```
void draw() {
  background(204);
  manageButtons();
  noStroke();
  fill(0);
  if (mode == 1) {
    ellipse(0, 40, 60, 60);
  } else if (mode == 2) {
    ellipse(50, 40, 60, 60);
  } else if (mode == 3) {
    ellipse(100, 40, 60, 60);
  }
}

void manageButtons() {
  button1.update();
  button2.update();
  button3.update();
  button1.display();
  button2.display();
  button3.display();
}

void mousePressed() {
  if (button1.press() == true) { mode = 1; }
  if (button2.press() == true) { mode = 2; }
  if (button3.press() == true) { mode = 3; }
}

void mouseReleased() {
  button1.release();
  button2.release();
  button3.release();
}
```

Using the same technique for rolling over a circle, it is straightforward to make a circular button. It's also possible to make a button with an irregular shape, but this requires a different technique.

Check boxes, Radio buttons

Check boxes and radio buttons form another category of interface elements. They frequently appear in windows for configuring a computer or for filling out forms on the

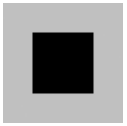
Web. The appearance of these elements is less important than their behavior, but check boxes are usually square and radio buttons are typically circular. Check boxes operate independently from others, while the status of a radio button is dependent on that of the others in its group. Thus check boxes are used as an interface element when more than one option is available, and radio buttons are used when only one element within a list of options can be selected.

A check box is a button with two states, ON and OFF, that change when the element is selected. If the current state is OFF and the box is selected it changes its state to ON, and vice versa. The `Check` class presented below defines the form and behavior of a check box. The fields and methods are similar to those in the `Button` class, but the `display()` method is unique. When the `press` field is `true`, an X is drawn in the center of the box to show that the state is ON.

47-08

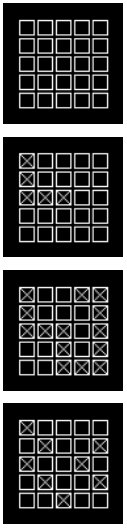
```
class Check {
  int x, y;                 // The x- and y-coordinates
  int size;                 // Dimension (width and height)
  color baseGray;           // Default gray value
  boolean checked = false;  // True when the check box is selected

  Check(int xp, int yp, int s, color b) {
    x = xp;
    y = yp;
    size = s;
    baseGray = b;
  }

  // Updates the boolean variable checked
  void press(float mx, float my) {
    if ((mx >= x) && (mx <= x+size) && (my >= y) && (my <= y+size)) {
      checked = !checked;  // Toggle the check box on and off
    }
  }

  // Draws the box and an X inside if the checked variable is true
  void display() {
    stroke(255);
    fill(baseGray);
    rect(x, y, size, size);
    if (checked == true) {
      line(x, y, x+size, y+size);
      line(x+size, y, x, y+size);
    }
  }
}
```

When a `Check` object is used in a program, the `display()` method should be run from `draw()` and the `press()` method should be run from `mousePressed()`. The next example presents one `Check` object in the center of the screen to show its behavior, and the example after that creates an array of `Check` objects and sets their positions to make a matrix.



47-09

```
// Requires the Check class

Check check;

void setup() {
  size(100, 100);
  // Inputs: x, y, size, fill color
  check = new Check(25, 25, 50, color(0));
}

void draw() {
  background(204);
  check.display();
}

void mousePressed() {
  check.press(mouseX, mouseY);
}
```



47-10

```
// Requires the Check class

int numChecks = 25;
Check[] checks = new Check[numChecks];

void setup() {
  size(100, 100);
  int x = 14;
  int y = 14;
  for (int i = 0; i < numChecks; i++) {
    checks[i] = new Check(x, y, 12, color(0));
    x += 15;
    if (x > 80) {
      x = 14;
      y += 15;
    }
  }
}
```

47-10 cont.

```
void draw() {
  background(0);
  for (int i=0; i<numChecks; i++) {
    checks[i].display();
  }
}

void mousePressed() {
  for (int i = 0; i < numChecks; i++) {
      checks[i].press(mouseX, mouseY);
  }
}
```

Like a check box, a radio button also has two states, ON and OFF. Unlike check boxes, radio buttons are used in groups of two or more, and only one element in the group can be selected at a time. Each radio button must be able to turn the other elements OFF when it's selected. The `Radio` class is unique from all other classes presented in the book because it is designed to use itself as one of its parameters. This makes it possible for each element within the `Radio` class array to reference the other elements in the array.

Look at this parameter in the `Radio` constructor on line 10 of code 47-11. The data type of the input is an array of `Radio` objects. The array is passed through the constructor and is then assigned to the `others[]` array, which also has the data type of an array of `Radio` objects. Inside the `press()` method, each `Radio` object sets the state of all the other `Radio` objects in the array to OFF if it has been clicked (turned ON). If the state for the object is ON, a black interior circle is drawn within the `display()` method.

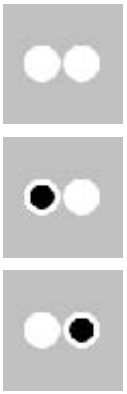
47-11

```
class Radio {
  int x, y;                 // The x- and y-coordinates of the rect
  int size, dotSize;        // Dimension of circle, inner circle
  color baseGray, dotGray;  // Circle gray value, inner gray value
  boolean checked = false;  // True when the button is selected
  int me;                   // ID number for this Radio object
  Radio[] others;           // Array of all other Radio objects

  Radio(int xp, int yp, int s, color b, color d, int m, Radio[] o) {
    x = xp;
    y = yp;
    size = s;
    dotSize = size - size/3;;
    baseGray = b;
    dotGray = d;
    others = o;
    me = m;
  }
```

47-11 cont.

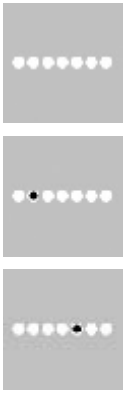
```
  // Updates the boolean value press, returns true or false
  boolean press(float mx, float my) {
    if (dist(x, y, mx, my) < size/2) {
      checked = true;
      for (int i = 0; i < others.length; i++) {
        if (i != me) {
          others[i].checked = false;
        }
      }
      return true;
    } else {
      return false;
    }
  }

  // Draws the element to the display window
  void display() {
    noStroke();
    fill(baseGray);
    ellipse(x, y, size, size);
    if (checked == true) {
      fill(dotGray);
      ellipse(x, y, dotSize, dotSize);
    }
  }
}
```

As mentioned, the most notable aspect of this example is the ability of the `Radio` objects to access the other members of their array. Notice the call to the constructor in code 47-12 below. The last parameter is the name of the array of radio objects. The `Radio` object's `display()` method is run from `draw()`, and the `press()` method is run from `mousePressed()`. The next example presents two `Radio` objects, and the example after it demonstrates the use of a `for` structure to iterate through a larger number of objects.



47-12

```
// Requires the Radio class

Radio[] buttons = new Radio[2];

void setup() {
  size(100, 100);
  smooth();
  // Inputs: x, y, size, base color, fill color,
  //         id number, array of others
  buttons[0] = new Radio(33, 50, 30, color(255), color(0),
                         0, buttons);
  buttons[1] = new Radio(66, 50, 30, color(255), color(0),
                         1, buttons);
}

void draw() {
  background(204);
  buttons[0].display();
  buttons[1].display();
}

void mousePressed() {
  buttons[0].press(mouseX, mouseY);
  buttons[1].press(mouseX, mouseY);
}
```



47-13

```
// Requires the Radio class

int numButtons = 7;
Radio[] buttons = new Radio[numButtons];

void setup() {
  size(100, 100);
  smooth();
  for (int i = 0; i < buttons.length; i++) {
    int x = i*12 + 14;
    buttons[i] = new Radio(x, 50, 10, color(255), color(0),
                           i, buttons);
  }
}

void draw() {
  background(204);
```

```
    for (int i = 0; i < buttons.length; i++) {
      buttons[i].display();
    }
  }

  void mousePressed() {
    for (int i = 0; i < buttons.length; i++) {
      buttons[i].press(mouseX, mouseY);
    }
  }
```
47-13 cont.

Scrollbar

A scrollbar is an interface element for selecting a value from a range of possible values. It can move through long lists of information such as Web pages and text documents. Scrollbars are typically narrow rectangular elements with a positionable interior element called a thumb. The user can move the thumb between the endpoints of the scrollbar by dragging it to a new position. The minimal `Scrollbar` class presented below creates horizontal scrollbars. There are multiple fields and methods for this class, but these values and their behavior will be familiar from the previous examples.

The `Scrollbar` class is similar to the other classes in the unit, but the `locked` field is unique, making it possible for the cursor to continue to update the position of the thumb even if the cursor moves off the scrollbar area. This common GUI feature lets people be less precise when moving the thumb element. If the mouse is pressed when the cursor is over the scrollbar, moving the mouse off the bar will continue to update the scrollbar until released. This class was designed so that each `Scrollbar` object has its own minimum and maximum values, defined by the parameters to the constructor. The `getPos()` method returns the current value of the thumb element within the scrollbar's range as defined by the `minVal` and `maxVal` fields.

```
class Scrollbar {                                                    47-14
  int x, y;              // The x- and y-coordinates
  float sw, sh;          // Width and height of scrollbar
  float pos;             // Position of thumb
  float posMin, posMax;  // Max and min values of thumb
  boolean rollover;      // True when the mouse is over
  boolean locked;        // True when its the active scrollbar
  float minVal, maxVal;  // Min and max values for the thumb

  Scrollbar (int xp, int yp, int w, int h, float miv, float mav) {
    x = xp;
    y = yp;
    sw = w;
```

```
  sh = h;
  minVal = miv;
  maxVal = mav;
  pos = x + sw/2 - sh/2;
  posMin = x;
  posMax = x + sw - sh;
}

// Updates the over boolean and the position of the thumb
void update(int mx, int my) {
  if (over(mx, my) == true) {
    rollover = true;
  } else {
    rollover = false;
  }
  if (locked == true) {
    pos = constrain(mx-sh/2, posMin, posMax);
  }
}

// Locks the thumb so the mouse can move off and still update
void press(int mx, int my) {
  if (rollover == true) {
    locked = true;
  } else {
    locked = false;
  }
}

// Resets the scrollbar to neutral
void release() {
  locked = false;
}

// Returns true if the cursor is over the scrollbar
boolean over(int mx, int my) {
  if ((mx > x) && (mx < x+sw) && (my > y) && (my < y+sh)) {
    return true;
  } else {
    return false;
  }
}
```

47-14 cont.

```
  // Draws the scrollbar to the screen
  void display() {
    fill(255);
    rect(x, y, sw, sh);
    if ((rollover==true) || (locked==true)) {
      fill(0);
    } else {
      fill(102);
    }
    rect(pos, y, sh, sh);
  }

  // Returns the current value of the thumb
  float getPos() {
    float scalar = sw/(sw-sh);
    float ratio = (pos - x) * scalar;
    float offset = minVal + (ratio/sw * (maxVal-minVal));
    return offset;
  }
}
```

The `Scrollbar` class is the longest code we've presented in this book, but integrating it into a program takes a single step. Like the other GUI elements in this unit, a `Scrollbar` object is declared at the top of the code, created in `setup()`, displayed and updated in `draw()`, and controlled by the mouse events `mousePressed()` and `mouseReleased()`. The next example features a pair of scrollbars with the same x-coordinates but different ranges. The top scrollbar selects integer numbers between 0 and 100, and the bottom scrollbar selects floating-point numbers between 0.0 and 1.0.

47-15

```
// Requires Scrollbar class

Scrollbar bar1, bar2;
PFont font;

void setup() {
  size(100, 100);
  noStroke();
  // Inputs: x, y, width, height, minVal, maxVal
  bar1 = new Scrollbar(10, 35, 80, 10, 0.0, 100.0);
  bar2 = new Scrollbar(10, 55, 80, 10, 0.0, 1.0);
  font = loadFont("Courier-30.vlw");
  textFont(font);
  textAlign(CENTER);
}
```

47-15 cont.

```
void draw() {
  background(204);
  fill(0);
  int pos1 = int(bar1.getPos());
  text(nf(pos1, 2), 50, 30);
  float pos2 = bar2.getPos();
  text(nf(pos2, 1, 2), 50, 90);
  bar1.update(mouseX, mouseY);
  bar2.update(mouseX, mouseY);
  bar1.display();
  bar2.display();
}

void mousePressed() {
  bar1.press(mouseX, mouseY);
  bar2.press(mouseX, mouseY);
}

void mouseReleased() {
  bar1.release();
  bar2.release();
}
```

The number returned from the scrollbar's `getPos()` method can be used to control any aspect of a program. The following example uses this number to set the position of an image.

47-16

```
// Requires Scrollbar Class

Scrollbar bar;
PImage img;

void setup() {
  size(100, 100);
  noStroke();
  // Inputs: x, y, width, height, minVal, maxVal
  bar = new Scrollbar(10, 45, 80, 10, -200.0, 0.0);
  img = loadImage("landscape.jpg");
}

void draw() {
  background(204);
  int x = int(bar.getPos());
  image(img, x, 0);
```

```
  bar.update(mouseX, mouseY);
  bar.display();
}

void mousePressed() {
  bar.press(mouseX, mouseY);
}

void mouseReleased() {
  bar.release();
}
```

47-16 cont.

Exercises

1. *Modify the `Button` class to work with circles.*
2. *Create a composition with check boxes and radio buttons.*
3. *Extend the `Scrollbar` class to have arrow buttons on the left and right that move the thumb one step each time an arrow is pressed.*

Structure 5: Objects II

This unit extends the discussion of object-oriented programming and introduces splitting a program into multiple constructors, composite objects, and inheritance.

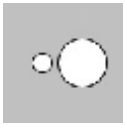
Syntax introduced:
`extends, super`

There is far more to object-oriented programming than was described in Structure 4 (p. 395). As your programs become longer and your ideas grow more ambitious, the additional object-oriented programming concepts and techniques discussed in this unit become important for managing code.

Multiple constructors

A class can have multiple constructors that assign the fields in different ways. Sometimes it's beneficial to specify every aspect of an object's data by assigning parameters to the fields, but other times it might be appropriate to define only one or a few.

In the next example, one constructor sets the x-coordinate, y-coordinate, and radius, while the other uses preset values. When the object is created, the program chooses the constructor to use depending on the number and type of variables specified. At the end of `setup()`, the `sp1` object is created using the first version of the `Spot` constructor, and the `sp2` object is created using the second version.



48-01

```
Spot sp1, sp2;

void setup() {
  size(100, 100);
  smooth();
  noLoop();
  // Run the constructor without parameters
  sp1 = new Spot();
  // Run the constructor with three parameters
  sp2 = new Spot(66, 50, 20);
}

void draw() {
  sp1.display();
  sp2.display();
}
```

48-01
cont.

```
class Spot {
  float x, y, radius;

  // First version of the Spot constructor;
  // the fields are assigned default values
  Spot() {
    x = 33;
    y = 50;
    radius = 8;
  }

  // Second version of the Spot constructor;
  // the fields are assigned with parameters
  Spot(float xpos, float ypos, float r) {
    x = xpos;
    y = ypos;
    radius = r;
  }

  void display() {
    ellipse(x, y, radius*2, radius*2);
  }
}
```

Composite objects

An object can include several other objects. Creating such composite objects is a good way to use the principles of modularity and build higher levels of abstraction. In the natural world, objects often possess components that operate independently but in relation to other components. Using a biological analogy, you might create a cell class, groups of which can be combined into muscle tissue and nervous tissue. These tissues can be combined into organs, and the organs into an organism. With multiple layers of abstraction, each step is built from composites of the previous layer. A bicycle class provides a different sort of example. It could be composed of objects for its frame, wheels, brakes, drivetrain, etc., and each of these units could be built from other classes. For example, the drivetrain could be built from objects for the pedals, crankset, and gears.

The following program combines the `Egg` class (p. 405) and the `Ring` class (p. 408) to create a new class called `EggRing`. It has one `Egg` object called `ovoid`, created in the constructor, and one `Ring` object called `circle`, created at the base of the class. The `transmit()` method calls the methods for both classes and resets `circle` when the object reaches its maximum size. As in all the examples using classes, the referenced classes have to be included in the sketch for the project to run.

```
// Requires Egg and Ring classes (codes 43-08 and 43-11)
```

48-02

```
class EggRing {
  Egg ovoid;
  Ring circle = new Ring();

  EggRing(int x, int y, float t, float sp) {
    ovoid = new Egg(x, y, t, sp);
    circle.start(x, y - sp/2);
  }

  void transmit() {
    ovoid.wobble();
    ovoid.display();
    circle.grow();
    circle.display();
    if (circle.on == false) {
      circle.on = true;
    }
  }
}
```

When the `EggRing` class is used in a program, each instance draws an egg to the screen with one `Ring` object growing from its center.

48-03

```
// Requires the Egg, Ring, and EggRing classes

EggRing er1, er2;

void setup() {
  size(100, 100);
  smooth();
  er1 = new EggRing(33, 66, 0.1, 33);
  er2 = new EggRing(66, 90, 0.05, 66);
}

void draw() {
  background(0);
  er1.transmit();
  er2.transmit();
}
```

Inheritance

A class can be defined using another class as a foundation. In object-oriented programming terminology, one class can *inherit* fields and methods from another. An object that *inherits* from another is called a *subclass*, and the object it inherits from is called a *superclass*. A subclass extends the superclass. When one class *extends* another, all of the methods and fields from the superclass are automatically included in the subclass. New fields and methods can be added to the subclass to build on the data and behavior of its superclass. If a method name is repeated within the subclass and has the same prototype (the same number of parameters with the same data types) as the one in the superclass, the method in the subclass overrides the other, thereby replacing it. When a method or field from the superclass is called from within the subclass, the name is prefaced with the keyword `super` to let the software know this method or field is a part of the superclass. The following examples clarify these new terms and concepts.

The `Spin` class was created to help explain the concept of inheritance. This very minimal class has fields for setting the x-coordinate, y-coordinate, speed, and angle. It has one method to update the angle.

48-04

```
class Spin {
  float x, y, speed;
  float angle = 0.0;

  Spin(float xpos, float ypos, float s) {
    x = xpos;
    y = ypos;
    speed = s;
  }

  void update() {
    angle += speed;
  }
}
```

The `SpinArm` class inherits elements from `Spin` and draws a line using the superclass's data. The constructor for `SpinArm` simply calls the constructor of the superclass. The `display()` function uses the inherited `x`, `y`, `angle`, and `speed` fields to draw a rotating line. Notice that the declarations for these fields are not repeated in the subclass because they are accessible to the subclass.

In the `SpinArm` constructor, `super()` is used to call the constructor of the `Spin` superclass. If `super()` with parameters is not used in the constructor of a subclass, a line calling `super()` with no parameters will be inserted behind the scenes. For this reason, any class meant to be extended will usually require a version of its constructor with no parameters, except in cases like this example where all subclasses call `super()` explicitly.

```
class SpinArm extends Spin {                                    48-05

  SpinArm(float x, float y, float s) {
    super(x, y, s);
  }

  void display() {
    strokeWeight(1);
    stroke(0);
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    line(0, 0, 100, 0);
    popMatrix();
  }
}
```

The `SpinSpots` class also inherits the elements of `Spin`. Like the `SpinArm` class, it uses its superclass's fields and constructor, but it builds even further on `Spin` by adding another field. The `dim` field was added to give the option to change the size of the circles. Notice that this field is declared at the top of the class, assigned in the constructor, and accessed in the `display()` method to set the size of the circles.

```
class SpinSpots extends Spin {                                  48-06
  float dim;

  SpinSpots(float x, float y, float s, float d) {
    super(x, y, s);
    dim = d;
  }

  void display() {
    noStroke();
    pushMatrix();
    translate(x, y);
    angle += speed;
    rotate(angle);
    ellipse(-dim/2, 0, dim, dim);
    ellipse(dim/2, 0, dim, dim);
    popMatrix();
  }
}
```

The process of creating objects from a subclass is identical to that of creating objects from a normal class. The class is the data type, and object variables of this type are declared, created, and accessed with the dot operator. In the following program, one `SpinSpot` object and one `SpinArm` object are declared and created. Their `update()` methods are used to increment the angle and draw to the screen. The class definitions for `Spin`, `SpinSpots`, and `SpinArm` must be included in the same page or put in separate tabs within the same sketch.

48-07

```
// Requires the Spin, SpinSpots, and SpinArm class

SpinSpots spots;
SpinArm arm;

void setup() {
  size(100, 100);
  smooth();
  arm = new SpinArm(width/2, height/2, 0.01);
  spots = new SpinSpots(width/2, height/2, -0.02, 33.0);
}

void draw() {
  background(204);
  arm.update();
  arm.display();
  spots.update();
  spots.display();
}
```

The next example extends the `Button` class (p. 439) introduced in Input 7. The extended class allows the cursor to move the button to different positions on the screen. This is one of the primary actions of most computer interfaces. The `DragButton` class inherits the behavior of responding to mouse events and extends this with the ability to move when it is clicked and dragged by the mouse. This subclass uses the existing `update()` and `display()` methods, augments the `press()` method, and adds a `drag()` method to update its position when the mouse is dragged.

48-08

```
class DragButton extends Button {
  int xoff, yoff;

  DragButton(int x, int y, int s, color bv, color ov, color pv) {
    super(x, y, s, bv, ov, pv);
  }

  void press(int mx, int my) {
```

```
    super.press();
    xoff = mx - x;
    yoff = my - y;
  }

  void drag(int mx, int my) {
    if (press == true) {
      x = mx - xoff;
      y = my - yoff;
    }
  }
}
```

48-08 cont.

The following example shows this new `DragButton` class embedded into a program. Its methods are run from the mouse event functions to register the status of the mouse with the `icon` object.

48-09

```
// Requires DragButton and Button classes

DragButton icon;

void setup() {
  size(100, 100);
  smooth();
  color gray = color(204);
  color white = color(255);
  color black = color(0);
  icon = new DragButton(21, 42, 50, gray, white, black);
}

void draw() {
  background(204);
  icon.update(mouseX, mouseY);
  icon.display();
}

void mousePressed()  { icon.press(mouseX, mouseY); }
void mouseReleased() { icon.release(); }
void mouseDragged()  { icon.drag(mouseX, mouseY); }
```

The `DragButton` class can be extended further to allow an image to be loaded and displayed as the icon. This class is very similar to `DragButton` but adds a field for the image and completely overrides the `display()` method to draw an outline around the image. This action provides visual feedback when the cursor is over the icon and when

the mouse is over the icon and pressed. Try integrating this new `DragImage` class into the previous example.

48-10

```
class DragImage extends DragButton {
  PImage img;

  DragImage(int x, int y, int d, String s) {
    super(x, y, d, color(204), color(255), color(0));
    img = loadImage(s);
  }

  // Override the display() from Button
  void display() {
    if (press == true) {
      stroke(pressGray);
    } else if (over == true) {
      stroke(overGray);
    } else {
      stroke(baseGray);
    }
    noFill();
    rect(x-1, y-1, size+1, size+1);
    image(img, x, y, size, size);
  }
}
```

While modular programming is an important technique, it can be too much of a good thing. Be careful to not abstract your code to a point where it becomes cumbersome. In the example above, the description of this simple button behavior with three classes is not practical. It was created this way to demonstrate key concepts of object-oriented programming, but could (and probably should) be simplified to one or two classes.

Exercises

1. *Write another constructor for the `Spot` class and use it within a variation of code 48-01.*
2. *Create your own composite class from two previously existing classes.*
3. *Create a unique subclass from the `Button` class.*

Simulate 1: Biology

This unit discusses the concept of software simulation and the topics of cellular automata and autonomous agents.

Simulation is used within physics, economics, the social sciences, and other fields to gain insight into the complicated systems that comprise our world. Software simulations employ the most powerful computers to model aspects of the world such as weather and traffic patterns. A tremendous amount of intellectual energy in the field of computer graphics has been dedicated to accurate simulation of lighting, textures, and the movement of physical materials such as cloth and hair. An entire genre of computer games exists that simulate city planning, military campaigns, and even everyday life. Computers constitute a powerful medium for simulating the processes of the world, and increasing computer speeds offer more sophisticated possibilities.

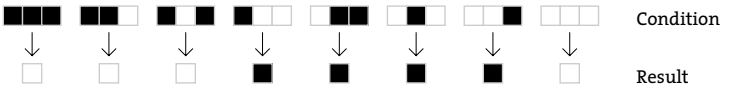
Within the arts, new technologies have often been used to represent and simulate nature. In ancient Greece, pneumatics animated sculptures. In the eighteenth century, precise gears provided the technical infrastructure for lifelike sculptures such as Vaucanson's Duck, which could "open its wings and flap them, while making a perfectly natural noise as if it were about to fly away."[1] In our contemporary world, computers and precision motors enable dancing robots and realistic children's toys that speak and move. One of the most fascinating simulations in recent art history is Wim Delvoye's *Cloaca* machine, which chemically and physically simulates the human digestive system.

Cellular automata

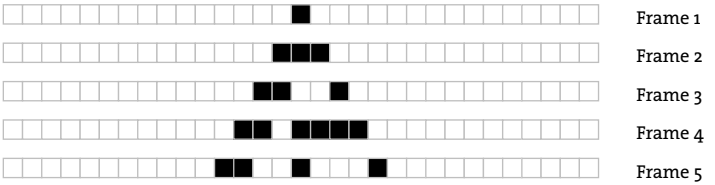
A cellular automaton (CA) is a self-operating system comprised of a grid of cells and rules stating how each cell behaves in relation to its neighbor. CAs were first considered by John von Neumann in the 1940s; they became well known in the 1970s after the publication of John Conway's Game of Life CA in a *Scientific American* article by Martin Gardner. CAs are intriguing because of their apparent simplicity in relation to the unexpected results they produce.

Steven Wolfram made important innovations in CA research in the early 1980s. Wolfram's one-dimensional CAs, each consisting of a single line of cells with rules, determine the value of each cell at each frame. The value of each cell is determined by its own value and those of its two neighbors. For example, if the current cell is white and its neighbors are black, it may also become black in the next frame. A set of rules determines when cells change their values. Since there are three cells with only two possible values (black or white), there are eight possible rules. In the diagram below, the three rectangles on the top are the three neighboring cells. The cell in the top middle is the current cell being evaluated, and those on the left and right are its neighbors.

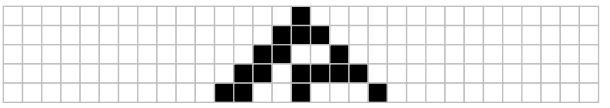
Depending on the current cell's value and that of its neighbors, the cell beneath the current cell is changed to black or white. The rules give rise to a number of possible variations and produce diverse results. One potential set of rules follows:



The CA starts with an initial state and is then updated to the next frame based on its rules. Each cell in the row is evaluated in relation to its two adjacent cells. Visual patterns begin to emerge from the minimal configuration of all white cells with one black cell in the middle:



The new one-dimensional image at each frame refers only to the previous frame. Because each frame is one-dimensional, it can be combined with the others to create a two-dimensional image, revealing the history of each frame of the CA within a single image that can be read from top to bottom:



The rules for this one-dimensional CA can be encoded as an array of 0s and 1s. Using the image at the top of this page as reference, if the configuration in the top row stays the same, the resulting bottom row can be defined as an array of 8 values, each a 1 or 0. This configuration can be coded as 0, 0, 0, 1, 1, 1, 1, 0 where 0 is white and 1 is black. Each of the other 256 possible configurations can be coded as a sequence of 8 numbers. Changing these numbers in the following example of a one-dimensional CA creates different images. The images on page 464 present some of the possible rules and their results.

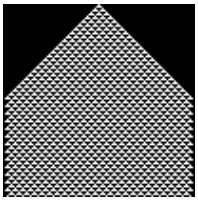
49-01

```
int[] rules = { 0, 0, 0, 1, 1, 1, 1, 0 };
int gen = 1;  // Generation
color on = color(255);
color off = color(0);

void setup() {
  size(101, 101);
  frameRate(8);  // Slow down to 8 frames each second
```

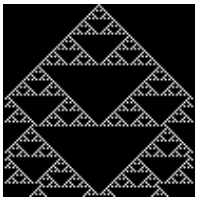
49-01 cont.

```
  background(0);
  set(width/2, 0, on);  // Set the top middle pixel to white
}

void draw() {
  // For each pixel, determine new state by examining current
  // state and neighbor states and ignore edges that have only
  // one neighbor
  for (int i = 1; i < width-1; i++) {
    int left = get(i-1, gen-1);         // Left neighbor
    int me = get(i, gen-1);             // Current pixel
    int right = get(i+1, gen-1);        // Right neighbor
    if (rules(left, me, right) == 1) {
      set(i, gen, on);
    }
  }
  gen++;                 // Increment the generation by 1
  if (gen > height-1) {  // If reached the bottom of the screen,
    noLoop();            // stop the program
  }
}

// Implement the rules
int rules(color a, color b, color c) {
  if ((a == on ) && (b == on ) && (c == on )) { return rules[0]; }
  if ((a == on ) && (b == on ) && (c == off)) { return rules[1]; }
  if ((a == on ) && (b == off) && (c == on )) { return rules[2]; }
  if ((a == on ) && (b == off) && (c == off)) { return rules[3]; }
  if ((a == off) && (b == on ) && (c == on )) { return rules[4]; }
  if ((a == off) && (b == on ) && (c == off)) { return rules[5]; }
  if ((a == off) && (b == off) && (c == on )) { return rules[6]; }
  if ((a == off) && (b == off) && (c == off)) { return rules[7]; }
  return 0;
}
```

John Conway's Game of Life predates Wolfram's discoveries by more than a decade. Gardner's article in *Scientific American* describes Conway's invention as "a fantastic solitaire pastime he calls 'life.' Because of its analogies with the rise, fall and alternations of a society of living organisms, it belongs to a growing class of what are called 'simulation games'—games that resemble real-life processes."[2] Life was not originally run with a computer, but early programmers rapidly became fascinated with it, and working with the Game of Life in software enabled new insight into the patterns that emerge as it runs.

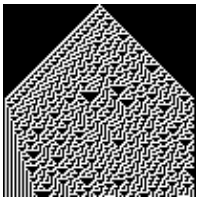


0,0,1,1,0,1,1,0

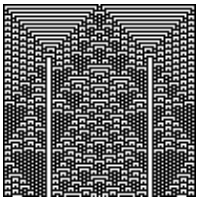


0,1,0,1,1,0,1,0

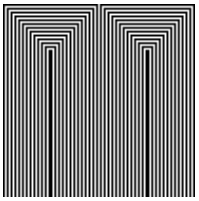
1,0,1,1,0,1,1,0



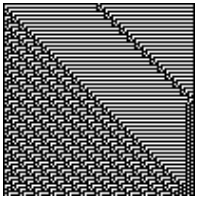
0,0,0,1,1,1,1,0



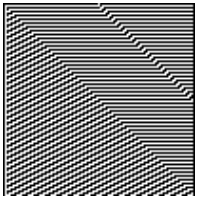
0,1,0,0,1,0,0,1



0,1,0,0,1,1,0,1

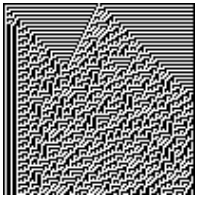


0,0,1,0,1,0,0,1



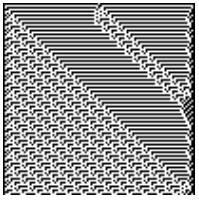
0,0,1,0,1,1,1,1

0,0,1,1,1,0,1,1

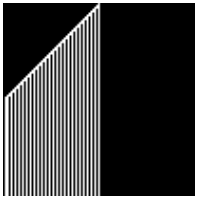


0,0,1,0,1,1,0,1

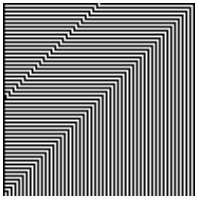
1,0,1,0,0,1,0,1



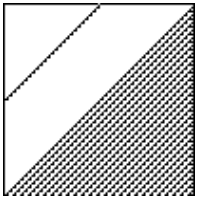
0,1,1,0,1,0,1,1



0,1,0,0,1,1,1,0



0,1,0,1,0,1,0,1



1,0,0,1,1,0,1,1

Wolfram's one-dimensional cellular automata

Use the numbers below each image as the data for the `rules[]` array in code 49-01 to watch each pattern generate.

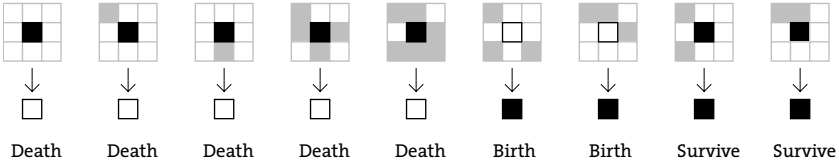
The Game of Life is a two-dimensional CA in which the rules for determining the value of each cell are defined by the neighboring cells in two dimensions. Each cell has eight neighboring cells, each of which can be named in relation to the directional orientation of the cell—north, northeast, east, southeast, south, southwest, west, northwest:

NW	N	NE
W		E
SW	S	SE

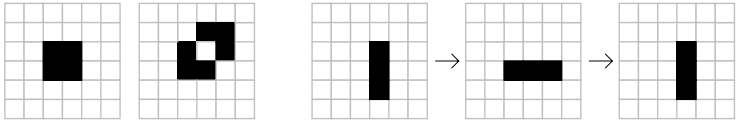
The rules for turning a cell on (alive) and off (dead) relate to the number of neighboring cells:

1. Death from isolation: Each live cell with less than two live neighbors dies in the next generation
2. Death from overpopulation: Each cell with four or more live neighbors dies in the next generation
3. Birth: Each dead cell with exactly three live neighbors comes to life in the next generation
4. Survival: Each live cell with two live neighbors survives in the next generation

Applying these rules to a cell reveals how different configurations translate into survival, death, and birth. In the image below, the cell being currently evaluated is in the center and is black if alive; neighbor cells are gray if alive and white if empty:

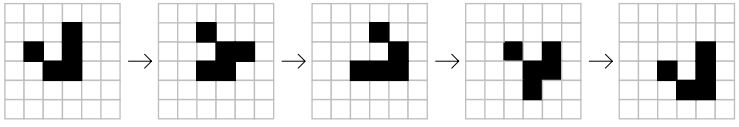


Different spatial configurations of cells create repeating patterns with each new generation. Some shapes are stable (do not change at each frame), some repeat, and some move across the screen:



Stable Configurations

Periodic Configuration



Moving object (this configuration moves one unit right and down over three generations)

Shapes called gliders are arrangements of cells that move across the grid, go through frames of physical distortion, and then arrive back at the same shape shifted by one grid unit. Repeating this pattern propels them across the grid.

The current state for the Game of Life is stored in a two-dimensional array of integers. A second grid hosts the next generation. At the end of each frame, the newly created generation becomes the old generation, and the process repeats. Cells are marked alive with the number 1 and and dead with 0. This makes it simple to count the number of neighbors by adding the values of neighboring cells. The `neighbors()` function looks at neighbors and counts the values of the adjacent cells. These numbers are used to set white or black pixels within `draw()`.

49-02

```
int[][] grid, futureGrid;

void setup() {
  size(540, 100);
  frameRate(8);
  grid = new int[width][height];
  futureGrid = new int[width][height];
  float density = 0.3 * width * height;
  for (int i = 0; i < density; i++) {
    grid[int(random(width))][int(random(height))] = 1;
  }
  background(0);
}

void draw() {
  for (int x = 1; x < width-1; x++) {
    for (int y = 1; y < height-1; y++) {
      // Check the number of neighbors (adjacent cells)
      int nb = neighbors(x, y);
      if ((grid[x][y] == 1) && (nb <  2)) {
        futureGrid[x][y] = 0;  // Isolation death
        set(x, y, color(0));
      } else if ((grid[x][y] == 1) && (nb >  3)) {
        futureGrid[x][y] = 0;  // Overpopulation death
        set(x, y, color(0));
      } else if ((grid[x][y] == 0) && (nb == 3)) {
        futureGrid[x][y] = 1;  // Birth
        set(x, y, color(255));
      } else {
        futureGrid[x][y] = grid[x][y];  // Survive
      }
    }
  }
```

```
  // Swap current and future grids
  int[][] temp = grid;
  grid = futureGrid;
  futureGrid = temp;
}

// Count the number of adjacent cells 'on'
int neighbors(int x, int y) {
  return grid[x][y-1] +    // North
         grid[x+1][y-1] +  // Northeast
         grid[x+1][y] +    // East
         grid[x+1][y+1] +  // Souteast
         grid[x][y+1] +    // South
         grid[x-1][y+1] +  // Southwest
         grid[x-1][y] +    // West
         grid[x-1][y-1];   // Northwest
}
```

49-02 cont.

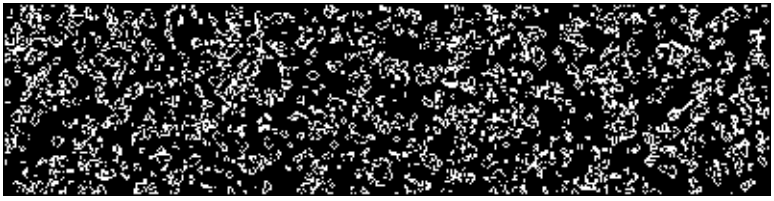
Changing the `neighbors()` function in code 49-02 to utilize the modulo operator (`%`) makes it possible for the cells to wrap from one side of the screen to the other. The `for` structures inside `draw()` also need to change to loop from `0` to `width` and `0` to `height`.

```
int neighbors(int x, int y) {
  int north = (y + height-1) % height;
  int south = (y + 1) % height;
  int east = (x + 1) % width;
  int west = (x + width-1) % width;
  return grid[x][north] +     // North
         grid[east][north] +  // Northeast
         grid[east][y] +      // East
         grid[east][south] +  // Southeast
         grid[x][south] +     // South
         grid[west][south] +  // Southwest
         grid[west][y] +      // West
         grid[west][north];   // Northwest
}
```

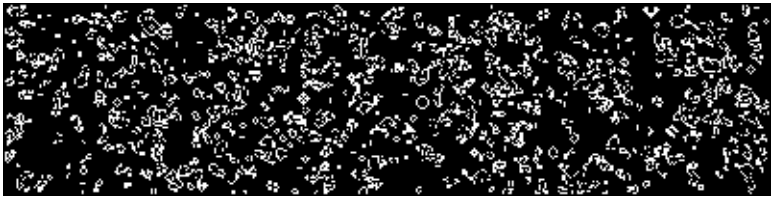
49-03

Research into cellular automata did not stop with Conway and Wolfram. Others have developed continuous CAs that are not limited to on/off states. Probabilistic CAs, for example, can partially or totally determine their rules through probabilities rather than absolutes. CAs possess the ability to simulate lifelike phenomena in spite of their basic format. For example, the patterns created with a one-dimensional CA can mimic patterns found in the shells of organisms such as cone snails. Other CAs produce images similar to those created by biochemical reactions.

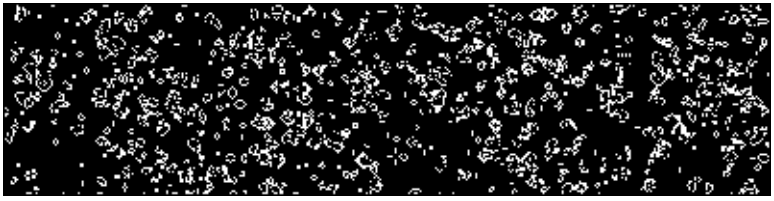
frameCount = 1



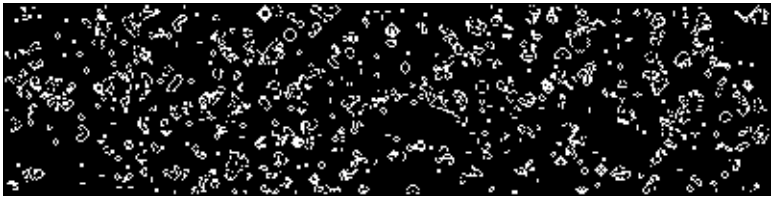
frameCount = 10



frameCount = 20



frameCount = 30



frameCount = 40

Conway's Game of Life

Using a few simple rules defined in code 49-02, the color relations between adjacent pixels create a dynamic ecosystem.

Autonomous agents

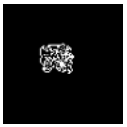
An autonomous agent is a system that senses and acts on its environment according to its own agenda. People, spiders, and plants are all autonomous agents. Each agent uses input from the environment as a basis for its actions. Each pursues it own goals, either consciously or through reflex. In his book *The Computational Beauty of Nature*, Gary William Flake defines an autonomous agent as "a unit that interacts with its environment (which probably consists of other agents) but acts independently from all other agents in that it does not take commands from some seen or unseen leader."[3] Agents aren't a part of a coordinated global plan, but structure does emerge from their interactions with other agents and the environment. The seemingly coordinated behavior of an ant colony and the order within a school of fish illustrate structured behavior emerging from the collective actions of individual agents.

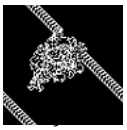
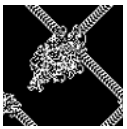
Like the examples of cellular automata presented above, autonomous agents can also exist in a grid world. Chris Langton's ant is a fascinating example. The ant can face only one of four directions: north, south, east, or west. Like cellular automata, the ant moves one frame at a time, behaving according to the following rules:

1. Move one frame forward
2. If on a white pixel, change the pixel to black and turn 90 degrees right
3. If on a black pixel, change the pixel to white and turn 90 degrees left

As the ant moves through the environment, it returns to the same pixel many times and each visit reverses the color of the pixel. Therefore, the future position of the ant is determined by its past movements. The remarkable thing about this ant is that with any starting orientation (north, south, east, or west), a sequence of actions that produce a straight path always emerges. From the seemingly chaotic mess upon which the ant embarks, an ordered path always develops. This program is not intended as a simulation of a real insect. It's an example of a software agent with extremely simple rules behaving in an entirely unexpected but ultimately predictable and structured manner. The instruction to eventually construct a straight path is never given, but it emerges through the rules of the ant in relation to its environment. The environment contains the memory of the ant's previous frames, which the ant uses to determine its next move.

In this example program, the ant's world wraps around from each edge of the screen to the opposite edge. Wrapping around to the other side of the screen, the ant is disrupted by its previous path. The order eventually emerges and the ant begins a new periodic sequence producing linear movement. In the code, directions are expressed as numbers. South is 0, east is 1, north is 2, and west is 3. At each frame, the ant moves one pixel forward based on its current orientation and then checks the color of the pixel at its location. It turns right by subtracting 1 and turns left by adding 1. Run the code to see the sequence change through time.



49-04

```
int SOUTH = 0;          // Direction numbers with names
int EAST = 1;           // so that the code self-documents
int NORTH = 2;
int WEST = 3;
int direction = NORTH;  // Current direction of the ant
int x, y;               // Ant's current position

color ON = color(255);  // Color for an 'on' pixel
color OFF = color(0);   // Color for an 'off' pixel

void setup() {
  size(100, 100);
  x = width/2;
  y = height/2;
  background(0);


void draw() {
  if (direction == SOUTH) {
    y++;
    if (y == height) {
      y = 0;
    }
  } else if (direction == EAST) {
    x++;
    if (x == width) {
      x = 0;
    }
  } else if (direction == NORTH) {
    if (y == 0) {
      y = height-1;
    } else {
      y--;
    }
  } else if (direction == WEST) {
    if (x == 0) {
      x = width-1;
    } else {
      x--;
    }
  }

  if (get(x, y) == ON) {
    set(x, y, OFF);
```

49-04
cont.

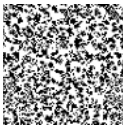
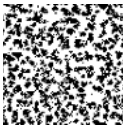
```
      if (direction == SOUTH) {
        direction = WEST;
      } else {
        direction--;
      }
    } else {
      set(x, y, ON);
      if (direction == WEST) {
        direction = SOUTH;
      } else {
        direction++;  // Rotate direction
      }
    }
  }
```

Mitchel Resnick's termite is another example that demonstrates order emerging from extremely simple rules. Like Langton's ant, this termite is not intended as a simulation of a real organism, but it exhibits remarkable behavior. The termite exists on a grid where a white unit represents open space and black represents a wood chip. The termite wanders through the space, and when it finds a wood chip it picks it up and wanders until it runs into another wood chip. Finding a wood chip causes it to drop its current load, turn around, and continue to wander. Over time, ordered piles emerge as a result of its effort.

In the code that creates the termite and its environment, the `angles[]` array contains the possible directions in which the termite can move. At each frame the termite moves one space on the grid. The angles specify which neighboring pixel it can move into:

NW −1,−1	N 0,−1	NE 1,−1
W −1,0		E 1,0
SW −1,1	S 0,1	SE 1,1

When space in front of the termite is open, it moves to the next space in the current direction or the next space in an adjacent direction. For example, if the current direction is south, it will move to the next space in the south, southeast, or southwest direction. If the current direction is northeast, it will move to the next space in the northeast, east, or north direction. When the termite does not have space in front and it's carrying a wood chip, it will reverse its direction and move one space in the new direction. When it does not have a space in front and it's not carrying a wood chip, it moves into the space occupied with the wood chip and picks it up.

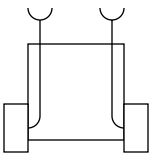



```
int[][] angles = {{ 0, 1 }, { 1, 1 }, { 1, 0 }, { 1,-1 },    49-05
                  { 0,-1 }, {-1,-1 }, {-1, 0 }, {-1, 1 }};

int numAngles = angles.length;
int x, y, nx, ny;
int dir = 0;
color black = color(0);
color white = color(255);

void setup() {
  size(100, 100);
  background(255);
  x = width/2;
  nx = x;
  y = height/2;
  ny = y;
  float woodDensity = width * height * 0.5;
  for (int i = 0; i < woodDensity; i++) {
    int rx = int(random(width));
    int ry = int(random(height));
    set(rx, ry, black);
  }
}

void draw() {
  int rand = int(abs(random(-1, 2)));
  dir = (dir + rand + numAngles) % numAngles;
  nx = (nx + angles[dir][0] + width) % width;
  ny = (ny + angles[dir][1] + height) % height;

  if ((get(x,y) == black) && (get(nx,ny) == white)) {
    // Move the chip one space
    set(x, y, white);
    set(nx, ny, black);
    x = nx;
    y = ny;
  } else if ((get(x,y) == black) && (get(nx,ny) == black)) {
    // Move in the opposite direction
    dir = (dir + (numAngles/2)) % numAngles;
    x = (x + angles[dir][0] + width) % width;
    y = (y + angles[dir][1] + height) % height;
  } else {
    // Not carrying
    x = nx;
```

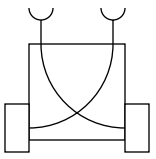
```
      y = ny;
    }
    nx = x;  // Save the current position
    ny = y;
  }
```

Other simulations of autonomous agents have been created without restrictive grids. These agents are allowed to move freely through their environment. Because they use floating-point numbers for position, they have more potential variations in location and orientation than the gridded CAs. Two of the best-known autonomous agents are Valentino Braitenberg's Vehicles and Craig Reynolds's Boids.

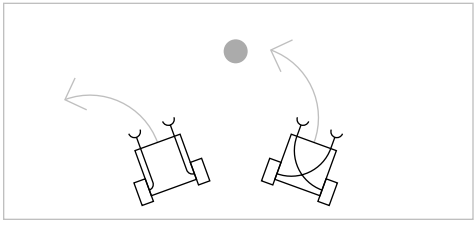
The neuroanatomist Valentino Braitenberg published *Vehicles: Experiments in Synthetic Psychology* in 1984. In this small, delightful book he presents conceptual schematics for fourteen unique synthetic creatures he calls Vehicles. Vehicle 1 has one sensor and one motor that are connected so that a strong stimulus will make the motor turn quickly and a weak stimulus will make the motor turn slowly. If the sensor registers nothing, the vehicle will not move. Vehicle 2 has two sensors and two motors. If they are correlated the same way as in Vehicle 1 they create Vehicle 2a and if they are crossed they create Vehicle 2b. If the sensor is attracted to light, for example, and there is a light in the room, Vehicle 2a will turn away from the light and Vehicle 2b will approach the light. Braitenberg characterizes these machines as correspondingly cowardly and aggressive to feature the anthropomorphic qualities we assign to moving objects:



Vehicle 2a



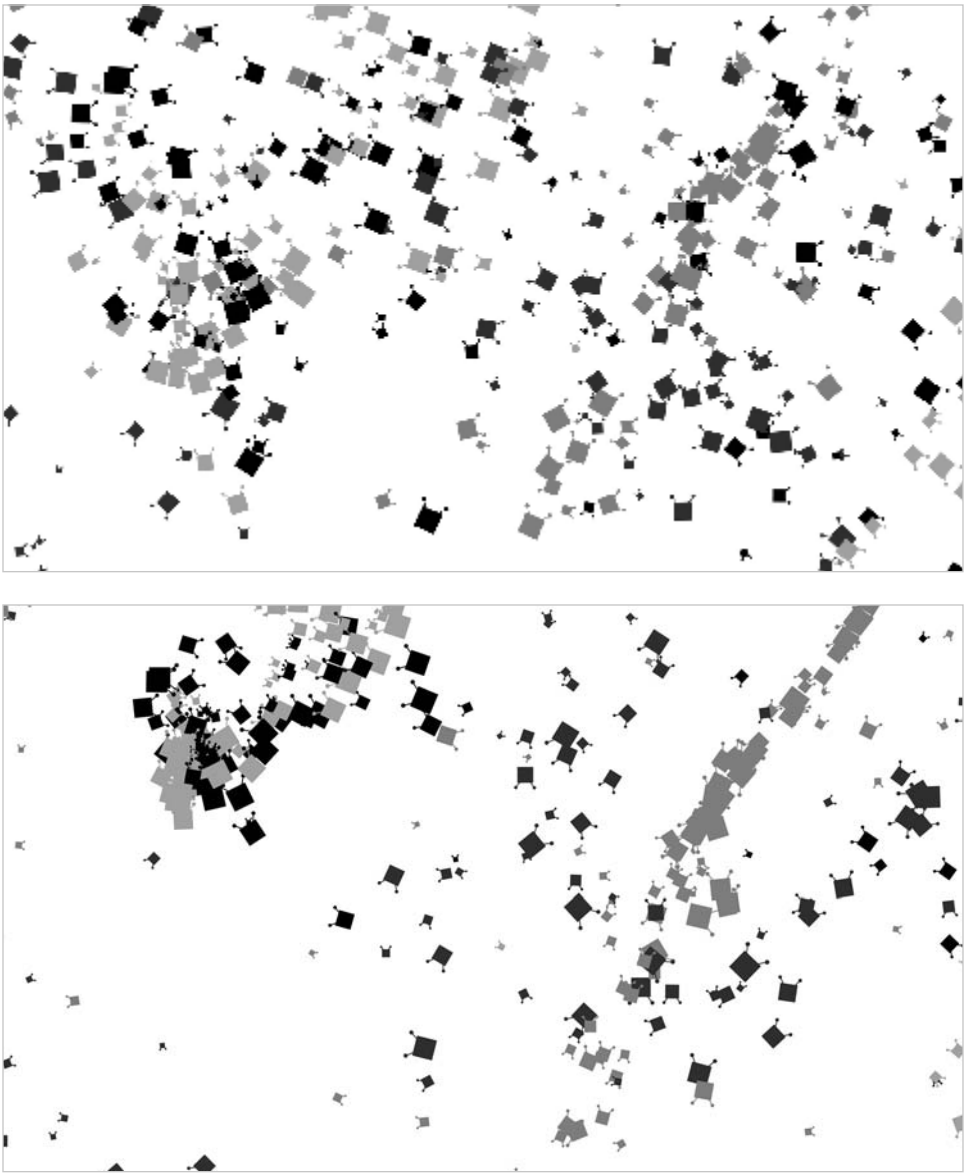
Vehicle 2b



Vehicle 2a and 2b movement in relation to a stimulus

Vehicle 3a and 3b are identical to Vehicle 2a and 2b but the correlation between the sensor and the motor is reversed—a weak sensor stimulus will cause the motor to turn quickly and a strong sensor stimulus causes the motors to stop. Vehicle 3a moves toward the light and stops when it gets too close, and 3b approaches the light but turns and leaves when it gets too close. If more than one stimulus is placed in the environment, these simple configurations can yield intricate paths of movement as they negotiate their attention between the competing stimuli.

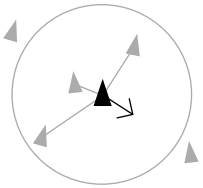
In 1986, Craig Reynolds developed the Boids software model to simulate coordinated animal motion like that of flocks of birds and schools of fish. To refute the common conception that these groups of creatures navigate by following a leader, Reynolds presented three simple behaviors that simulated a realistic flocking behavior without



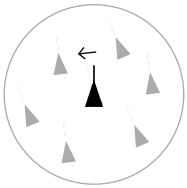
Braitenberg's Vehicles

Five hundred vehicles move through the environment. Each gray value represents a different category of vehicles. The vehicles in each category share the same behavior (follow the same rules), so over time they form groups.

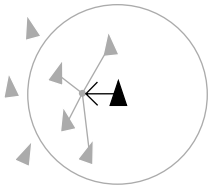
the need for a hierarchy. These behaviors define how each creature behaves in relation to its neighbors:



Separation:
Steer to avoid crowding local flockmates



Alignment:
Steer toward the average heading of local flockmates



Cohesion:
Steer to move toward the average position of local flockmates

The flocking rules provide an evocative example of emergence, the phenomenon of a global behavior originating from the interactions of simple, local behaviors. The flocking behavior of the group is not overtly programmed, but emerges through the interaction of each software unit based on the simple rules. The Pond example on page 497 is an implementation of Boids.

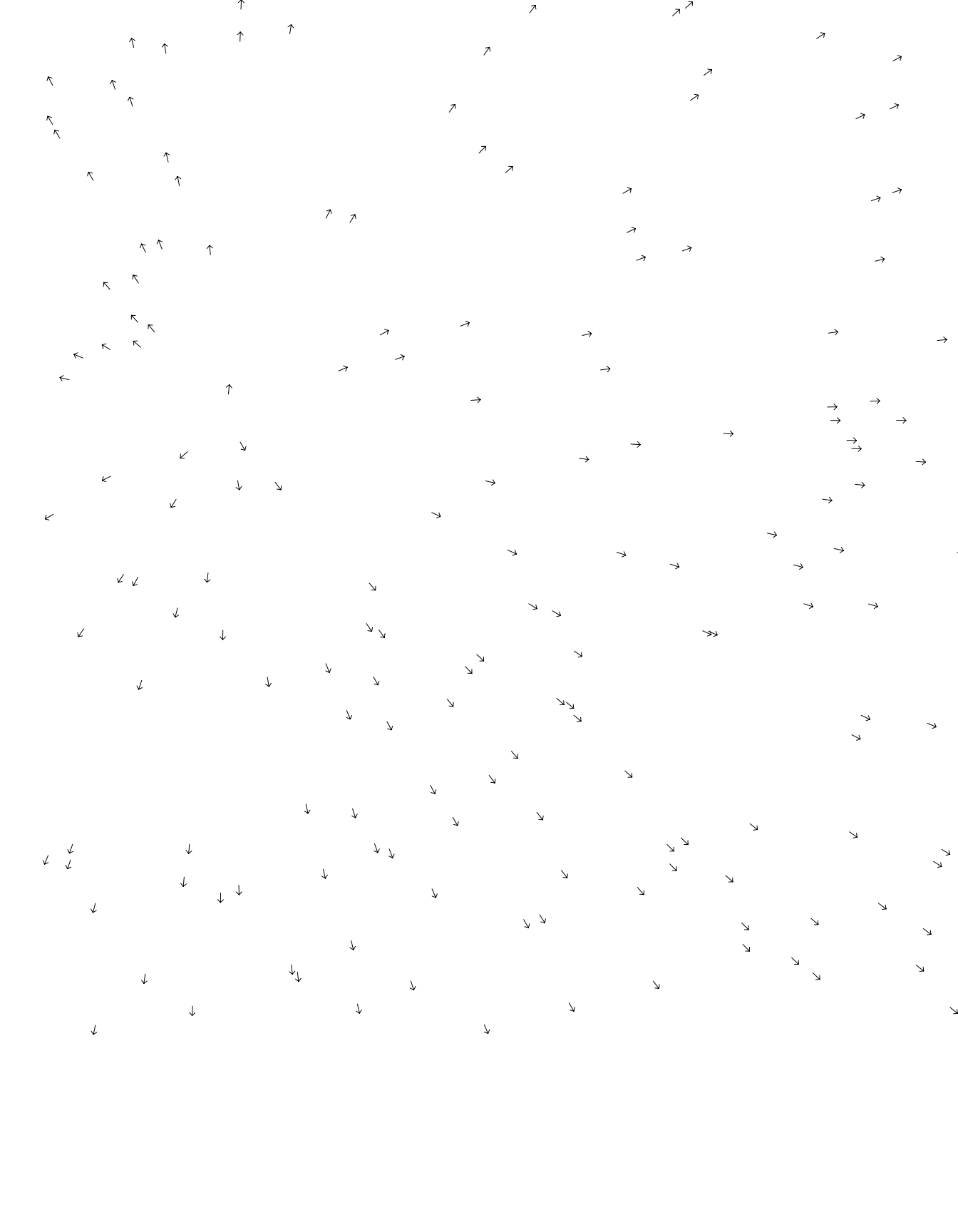
The autonomous agent simulations presented here were at the cutting edge of research over twenty years ago. They have since become some of the most popular examples for presenting the ideas of agency and emergence. The ideas from research in autonomous agents has been extended into many disciplines within art and science including sculpture, game design, and robotics.

Exercises

1. *Increase the size of the grid for Wolfram's one-dimensional CA. There are 256 possible rule sets for this one program and only 13 are presented in this unit. Try some of the other options. Which do you find the most interesting? Can the diverse results be put into categories?*
2. *Increase the size of the grid for Conway's Game of Life. Can you find other stable, periodic, or moving configurations?*
3. *Extend the termite code to have more than one termite moving chips of wood.*

Notes

1. Alfred Chapuis and Edmond Droz, *Automata: A Historical and Technological Study*, translated by Alec Reid (Editions du Griffon, 1958).
2. Martin Gardner, "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life,'" *Scientific American* 223 (October 1970), pp. 120 - 123.
3. Gary William Flake, *The Computational Beauty of Nature* (MIT Press, 1998), p. 261.



Simulate 2: Physics

This unit introduces physical simulations for particle systems and springs.

Physical simulation, a technique that creates relationships between software elements and the rules of the physical world, helps people relate to what they see on screen. A long list of technical papers and books have been written about this topic. Video game and 3D computer animation communities have devoted tremendous energy to simulating aspects of the world such as the collision of solid objects and physical phenomena such as fire and water. Because a discussion of physical simulation could occupy an entire book, this unit will only present some of the basic concepts. The domain of physical simulation is introduced through terminology and presented as two flexible simulation techniques, for particle systems and springs.

Motion simulation

To simulate physical phenomena in software, a mathematical model is required. Newtonian physics, developed circa 1687 by Isaac Newton, provides an adequate model based on velocity, acceleration, and friction. These terms are introduced and explained in sequence through the examples that follow.

The first example to generate motion (code 31-01, p. 279) uses a variable named *speed* to create movement. At each frame of animation, the variable named *y* is updated by the *speed* variable:

```
y = y + speed
```

Using this code, the position of a circle set by the variable `y` is changed by the same amount every frame. The code does not take into account other forces that might be exerted on the circle. For example, the circle might have a large mass, or gravity may apply a strong force, or it might be moving across a rough surface so that high friction slows it down. These forces are omnipresent in the physical world, but they affect a software simulation only if they are included as parts of the design. They need to be calculated at each frame to exert their influence.

Instead of relying solely on a variable representing speed to mimic realistic motion, variables are created to store the velocity and acceleration. The velocity changes the position of the element, and acceleration changes the velocity.

Velocity defines the speed and direction as one number. For example, a velocity of –5 moves the position in a negative direction at a speed of 5. A velocity of 12 moves the position in a positive direction at a speed of 12. Speed is defined as the magnitude (absolute value) of the velocity.

Acceleration defines the rate of change in the velocity. An acceleration value greater than zero means the velocity will increase each frame, and an acceleration value less than zero means the velocity will decrease each frame. Using the velocity and acceleration values to control the position of a visual element causes it to change direction and to increase or decrease its speed. The position of an object is updated with two steps:

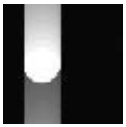
```
velocity = velocity + acceleration
y = y + velocity
```

The following example is similar to code 31-01, but uses `velocity` and `acceleration` variables instead of a single `speed` variable. Because the acceleration value is 0.01, the velocity increases, therefore moving the circle faster each frame.

50-01

```
float y = 50.0;
float radius = 15.0;
float velocity = 0.0;
float acceleration = 0.01;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 10);
  rect(0, 0, width, height);
  fill(255);
  ellipse(33, y, radius, radius);
  velocity += acceleration;  // Increase the velocity
  y += velocity;             // Update the position
  if (y > height+radius) {   // If over the bottom edge,
    y = -radius;             // move to the top
  }
}
```

In the following example, the circle continually slows down until it eventually stops and changes direction. This happens because the negative acceleration value gradually decreases the velocity until it becomes negative.



50-02

```
float y = 50.0;
float radius = 15.0;
float velocity = 9.0;
float acceleration = -0.05;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  ellipse(33, y, radius, radius);
  velocity += acceleration;
  y += velocity;
  if (y > height+radius) {
    y = -radius;
  }
}
```

Friction is a force that impacts velocity. The speed of a book pushed across a table is affected by friction between the two surfaces. A paper airplane is affected by the friction of the air. In code, friction is a number between 0.0 and 1.0 that decreases the velocity. In the next example, the friction value is multiplied by the velocity value each frame to gradually decrease the distance traveled by the circle each frame. Change the value of the `friction` variable in the code below to see its effect on the ball's movement.

50-03

```
float y = 50.0;
float radius = 15.0;
float velocity = 8.0;
float friction = 0.98;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
```

```
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  ellipse(33, y, radius, radius);
  velocity *= friction;
  y += velocity;
  if (y > height+radius) {
    y = -radius;
  }
}
```

50-03 cont.

The direction and speed components of the velocity can be altered independently. Reversing the direction of the velocity simulates bouncing. The following example inverts the velocity when the edge of the circle touches the bottom of the display window. The acceleration of 0.1 simulates gravity, and the friction gradually reduces the velocity to stop the bouncing eventually.

50-04

```
float x = 33;
float y = 5;
float velocity = 0.0;
float radius = 15.0;
float friction = 0.99;
float acceleration = 0.3;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  velocity += acceleration;
  velocity *= friction;
  y += velocity;
  if (y > (height-radius)) {
    y = height - radius;
    velocity = -velocity;
  }
  ellipse(x, y, radius, radius);
}
```

Particle systems

A particle system, an array of particles that responds to the environment or to other particles, serves to simulate and render phenomena such as fire, smoke, and dust. Hollywood films and video game companies frequently employ particle systems to create realistic explosions and water effects. Particles are affected by forces and are typically used to simulate physical laws for generating motion.

Writing a simple `Particle` class can help manage the complexity of a particle system. The `Particle` class has fields for the `radius` and `gravity` and pairs of fields to store the position and velocity. Gravity acts like the `acceleration` variable in previous examples. The class methods update the position and draw the particle to the display window. The parameters to the constructor set the initial position, velocity, and radius.

50-05

```
class Particle {
  float x, y;           // The x- and y-coordinates
  float vx, vy;         // The x- and y-velocities
  float radius;         // Particle radius
  float gravity = 0.1;

  Particle(int xpos, int ypos, float velx, float vely, float r) {
    x = xpos;
    y = ypos;
    vx = velx;
    vy = vely;
    radius = r;
  }

  void update() {
    vy = vy + gravity;
    y += vy;
    x += vx;
  }

  void display() {
    ellipse(x, y, radius*2, radius*2);
  }
}
```

The following example shows how to use the `Particle` class. Here, as in most examples that use objects, an object variable is declared outside of `setup()` and `draw()` and created within `setup()`, and its methods are run within `draw()`. The example throws a particle across the display window from the lower-left to the upper-right corner. After the particle moves off the screen, its values continue to update but it can no longer be seen.

50-06

```
// Requires Particle class

Particle p;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  p = new Particle(0, height, 2.2, -4.2, 20.0);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  p.update();
  p.display();
}
```

The `Particle` class is very limited, but allows the extension and creation of more applicable behavior. The `GenParticle` class extends the `Particle` class so the particle returns to its origin after it moves outside the display window. This allows for a continuous flow of particles with a fixed number of objects. In the example below, the two variables `originX` and `originY` store the coordinates of the origin, and the `regenerate()` method repositions the particle when it is outside the display window and resets its velocity.

50-07

```
class GenParticle extends Particle {
  float originX, originY;

  GenParticle(int xIn, int yIn, float vxIn, float vyIn,
              float r, float ox, float oy) {
    super(xIn, yIn, vxIn, vyIn, r);
    originX = ox;
    originY = oy;
  }

  void regenerate() {
    if ((x > width+radius) || (x < -radius) ||
        (y > height+radius) || (y < -radius)) {
      x = originX;
      y = originY;
      vx = random(-1.0, 1.0);
```

```
      vy = random(-4.0, -2.0);
    }
  }
}
```

50-07 cont.

The `GenParticle` object is used the same way as a `Particle` object, but the `regenerate()` method also needs to be run to ensure an endless supply of flowing particles. In the following example, 200 particles are created within an array and individually modified.

50-08

```
// Requires Particle and GenParticle classes

int numParticles = 200;
GenParticle[] p = new GenParticle[numParticles];

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  for (int i = 0; i < p.length; i++) {
    float velX = random(-1, 1);
    float velY = -i;
    // Inputs: x, y, x-velocity, y-velocity,
    //         radius, origin x, origin y
    p[i] = new GenParticle(width/2, height/2, velX, velY,
                           5.0, width/2, height/2);
  }
}

void draw() {
  fill(0, 36);
  rect(0, 0, width, height);
  fill(255, 60);
  for (int i = 0; i < p.length; i++) {
    p[i].update();
    p[i].regenerate();
    p[i].display();
  }
}
```

The `LimitedParticle` class extends the `Particle` class to change the direction of the velocity when a particle hits the bottom of the display window. It also introduces friction so the motion of each particle is reduced each frame.

50-09

```
class LimitedParticle extends Particle {
  float friction = 0.99;

  LimitedParticle(int ix, int iy, float ivx, float ivy, float ir) {
    super(ix, iy, ivx, ivy, ir);
  }

  void update() {
    vy *= friction;
    vx *= friction;
    super.update();
    limit();
  }

  void limit() {
    if (y > height-radius) {
      vy = -vy;
      y = constrain(y, -height*height, height-radius);
    }
    if ((x < radius) || (x > width-radius)) {
      vx = -vx;
      x = constrain(x, radius, width-radius);
    }
  }
}
```

The `LimitedParticle` class is used in the following examples to create a screen full of small bouncing elements. Each starts with a different velocity, but they all slow down and eventually come to rest at the bottom of the screen.

50-10

```
// Requires Particle and LimitedParticle classes

int num = 80;
LimitedParticle[] p = new LimitedParticle[num];
float radius = 1.2;

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  for (int i = 0; i < p.length; i++) {
    float velX = random(-2, 2);
    float velY = -i;
    // Inputs: x, y, x-velocity, y-velocity, radius
```

```
    p[i] = new LimitedParticle(width/2, height/2,
                               velX, velY, 2.2);
  }
}

void draw() {
  fill(0, 24);
  rect(0, 0, width, height);
  fill(255);
  for (int i = 0; i < p.length; i++) {
    p[i].update();
    p[i].display();
  }
}
```

50-10 cont.

The particles in the previous examples are drawn as circles to make the code simple to read. Particles can, however, be drawn as any shape. The following example makes another class from the original `Particle` class. This `ArrowParticle` class uses the fields and methods from its superclass to control the velocity and position of the particle, but it adds code to calculate an angle and to draw an arrow shape. The `atan2()` function is used to determine the current angle of the arrow. This value is used to set the rotation value in line 11. The arrow is positioned horizontally, but the rotation changes it to point up or down.

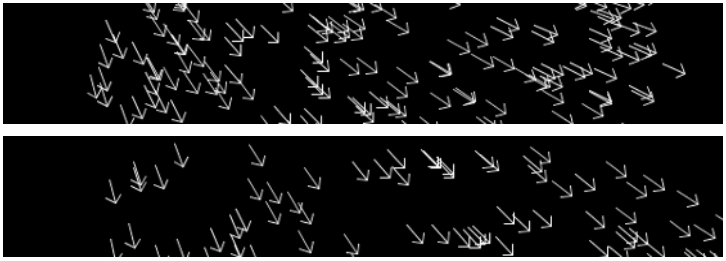
50-11

```
class ArrowParticle extends Particle {
  float angle = 0.0;
  float shaftLength = 20.0;

  ArrowParticle(int ix, int iy, float ivx, float ivy, float ir) {
    super(ix, iy, ivx, ivy, ir);
  }

  void update() {
    super.update();
    angle = atan2(vy, vx);
  }

  void display() {
    stroke(255);
    pushMatrix();
    translate(x, y);
    rotate(angle);
    scale(shaftLength);
    strokeWeight(1.0 / shaftLength);
```

```
    line(0, 0, 1, 0);
    line(1, 0, 0.7, -0.3);
    line(1, 0, 0.7, 0.3);
    popMatrix();
  }
}
```

50-11 cont.

Each arrow is assigned a random value within a range. The x-velocity ranges from 1.0 to 8.0 and the y-velocity from –5 to –1. At each frame, the force of gravity is applied to each particle and the angle of each arrow slowly turns toward the ground until it eventually disappears off the bottom of the screen.



50-12

```
// Requires Particle, ArrowParticle classes

int num = 320;
ArrowParticle[] p = new ArrowParticle[num];
float radius = 1.2;

void setup() {
  size(600, 100);
  smooth();
  for (int i = 0; i < p.length; i++) {
    float velX = random(1, 8);
    float velY = random(-5, -1);
    // Parameters: x, y, x-velocity, y-velocity, radius
    p[i] = new ArrowParticle(0, height/2, velX, velY, 1.2);
  }
}

void draw() {
  background(0);
  for (int i = 0; i < p.length; i++) {
    p[i].update();
    p[i].display();
  }
}
```

Springs

A spring is an elastic device, usually a coil of metal wire, that returns to its original shape after it has been extended or compressed. Software simulations of springs approximate the behavior of their physical analogs. The physics of a spring is simple and versatile. Like gravity, a spring is represented as a force. The force is calculated based on how "stiff" the spring is and how far it is stretched. The force of a spring is inversely proportional to how far it is stretched. This is known as Hooke's Law. The equation for calculating the force of a spring is:

$$f = -kx$$

In this equation, k is the spring stiffness constant, and the variable x is how far the spring is stretched. The inverse of k is multiplied by x to yield the force. The value of k is always between 0.0 and 1.0. This equation can be rewritten for clarity:

```
springForce = -stiffness * stretch
```

The `stretch` variable is the difference between the position and the target position:

```
springForce = -stiffness * (position - restPosition)
```

The equation can be simplified slightly to remove the negation:

```
springForce = stiffness * (restPosition - position)
```

A damping (frictional) force can be added to resist the motion. The value of the `damping` variable is always between 0.0 (so much friction that there is no movement) and 1.0 (no friction). The velocity of the spring is calculated by adding the spring force to the current velocity and then multiplying by the damping constant:

```
velocity = friction * (velocity + springForce)
```

These equations enable the writing of a simple spring simulation. The following example uses them to set the position of a rectangle. The `targetY` variable is the resting position and is the `y` variable is the current position of the spring. Try changing the values for the `stiffness` and `friction` variables to see how they affect the behavior of the rectangle. Both of these values should be in the range of 0.0 to 1.0.

50-13

```
float stiffness = 0.1;
float damping = 0.9;
float velocity = 0.0;
float targetY;
float y;

void setup() {
  size(100, 100);
  noStroke();
}
```

↓

50-13 cont.

```
void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  float force = stiffness * (targetY - y);  // f = -kx
  velocity = damping * (velocity + force);
  y += velocity;
  rect(10, y, width-20, 12);
  targetY = mouseY;
}
```

Mass is another component to simulate when working with springs. The mass of a spring affects how much effect a force will have. Commonly confused with weight, mass is the amount of matter an object consists of and is independent of the force of gravity; weight is the force applied by gravity on an object. If an object has more matter, gravity has a stronger effect on it and it therefore weighs more. Newton's second law states that the sum of the forces acting on an object is equal to the object's mass multiplied by the object's acceleration:

$$F = ma$$

This equation can be rearranged to solve for the acceleration:

$$a = F/m$$

Using code notation we've introduced, the equation can be written as follows:

```
acceleration = springForce / mass
```

This arrangement highlights the fact that an object with more mass will have less acceleration than one with less.

The following example is similar to code 50-13, but it positions two rectangles on the screen and therefore requires additional variables. It demonstrates the effect of mass on the spring equations. The rectangle on the right has a mass six times larger than that of the rectangle on the right.

50-14

```
float y1, y2;
float velocity1, velocity2;
float mass1 = 1.0;
float mass2 = 6.0;
float stiffness = 0.1;
float damping = 0.9;

void setup() {
  size(100, 100);
  noStroke();
}
```

```
void draw() {                                                    50-14
  fill(0, 12);                                                   cont.
  rect(0, 0, width, height);
  fill(255);

  float targetY = mouseY;

  float forceA = stiffness * (targetY - y1);
  float accelerationY1 = forceA / mass1;
  velocity1 = damping * (velocity1 + accelerationY1);
  y1 += velocity1;
  rect(10, y1, 40, 15);

  float forceB = stiffness * (targetY - y2);
  float accelerationY2 = forceB / mass2;
  velocity2 = damping * (velocity2 + accelerationY2);
  y2 += velocity2;
  rect(50, y2, 40, 15);
}
```

The `Spring2D` class encapsulates the concepts and equations from the previous examples into a reusable code unit. It calculates the spring values separately for the x- and y-axis and adds a gravitational force by combining the `gravity` value with the `forceY` value.

```
class Spring2D {                                                 50-15
  float vx, vy;   // The x- and y-axis velocities
  float x, y;     // The x- and y-coordinates
  float gravity;
  float mass;
  float radius = 10;
  float stiffness = 0.2;
  float damping = 0.7;

  Spring2D(float xpos, float ypos, float m, float g) {
    x = xpos;
    y = ypos;
    mass = m;
    gravity = g;
  }

  void update(float targetX, float targetY) {
    float forceX = (targetX - x) * stiffness;
    float ax = forceX / mass;
```

50-15 cont.

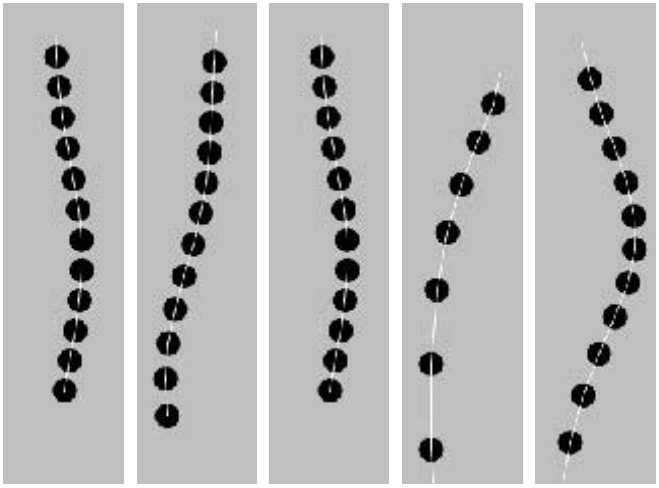
```
    vx = damping * (vx + ax);
    x += vx;
    float forceY = (targetY - y) * stiffness;
    forceY += gravity;
    float ay = forceY / mass;
    vy = damping * (vy + ay);
    y += vy;
  }

  void display(float nx, float ny) {
    noStroke();
    ellipse(x, y, radius*2, radius*2);
    stroke(255);
    line(x, y, nx, ny);
  }
}
```

The following example has two `Spring2D` objects. The position of the top element in the chain is controlled by the cursor, and the rest position of the bottom element is controlled by the position of the top. Try changing the value of the `gravity` variable to increase and decrease the space between the elements.

50-16

```
// Requires Spring2D Class

Spring2D s1, s2;
float gravity = 5.0;
float mass = 2.0;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
  // Inputs: x, y, mass, gravity
  s1 = new Spring2D(0.0, width/2, mass, gravity);
  s2 = new Spring2D(0.0, width/2, mass, gravity);
}

void draw() {
  background(204);
  s1.update(mouseX, mouseY);
  s1.display(mouseX, mouseY);
  s2.update(s1.x, s1.y);
  s2.display(s1.x, s1.y);
}
```

In the following example, an array is used to store more `Spring2D` objects. The rest position for each object is set by the object that immediately precedes it in the chain. The motion propagates through each element.



50-17

```
// Requires Spring2D Class

int numSprings = 30;
Spring2D[] s = new Spring2D[numSprings];
float gravity = 5.0;
float mass = 3.0;

void setup() {
  size(100, 900);
  smooth();
  fill(0);
  for (int i = 0; i < numSprings; i++) {
    s[i] = new Spring2D(width/2, i*(height/numSprings), mass, gravity);
  }
}

void draw() {
  background(204);
  s[0].update(mouseX, mouseY);
  s[0].display(mouseX, mouseY);
  for (int i = 1; i < numSprings; i++) {
    s[i].update(s[i-1].x, s[i-1].y);
    s[i].display(s[i-1].x, s[i-1].y);
  }
}
```

Metal springs each have a length to which they return after being pulled or pushed. In the spring simulations in this book, the final step is to give the spring a fixed length. This makes the springs more like their counterparts in the physical world. The `FixedSpring` class extends the `Spring2D` class to force the spring to have a specific length. The distance between the elements in the previous examples was created by a large gravitational force, but here the displacement is enforced by the `springLength` variable.

50-18

```
class FixedSpring extends Spring2D {
  float springLength;

  FixedSpring (float xpos, float ypos, float m, float g, float s) {
    super(xpos, ypos, m, g);
    springLength = s;
  }

  void update(float newX, float newY) {
    // Calculate the target position
    float dx = x - newX;
    float dy = y - newY;
    float angle = atan2(dy, dx);
    float targetX = newX + cos(angle) * springLength;
    float targetY = newY + sin(angle) * springLength;
    // Activate update method from Spring2D
    super.update(targetX, targetY);
    // Constrain to display window
    x = constrain(x, radius, width-radius);
    y = constrain(y, radius, height-radius);
  }
}
```

The `FixedSpring` class was written to extend the `Spring2D` class, but it also could have been written as its own class. It was written as a subclass to utilize the existing code, but this decision meant that the default values for the `stiffness` and `damping` fields introduced in `Spring2D` became the default values for `FixedSpring`. To avoid this restriction, the class can be modified to pass these values as parameters to the constructor. When creating a class you decide which fields to pass through the constructor by using your best judgment, but there is usually no single correct way to structure a program. There are many ways to write any program, and while the decisions about how to modularize the code should be made carefully, they can always be changed.

The following example calculates and draws one fixed-length spring to the display windows. Unlike the previous spring examples where the mass dangles from the cursor,

a fixed-length spring always tries to maintain its length. It can be balanced on top of the cursor as well as hung from the end.

50-19

```
// Requires Spring2D and FixedSpring classes

FixedSpring s;
float gravity = 0.5;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
  // Inputs: x, y, mass, gravity, length
  s = new FixedSpring(0.0, 50.0, 1.0, gravity, 40.0);
}

void draw() {
  background(204);
  s.update(mouseX, mouseY);
  s.display(mouseX, mouseY);
}
```

Fixed springs can be connected to other springs to create new forms. In the following example, two springs are joined so that the position of each spring affects the position of the other. When they are pushed too close together or pulled too far apart, they return to their defined distance from one another. Because a gravitational force is applied, they always fall to the bottom of the display window, but the force that keeps them apart is stronger, so they appear to move as a single, solid object.

50-20

```
// Requires Spring2D and FixedSpring classes

FixedSpring s1, s2;
float gravity = 1.2;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
  // Inputs: x, y, mass, gravity, length
  s1 = new FixedSpring(45, 33, 1.5, gravity, 40.0);
  s2 = new FixedSpring(55, 66, 1.5, gravity, 40.0);
}
```

```
void draw() {
  background(204);
  s1.update(s2.x, s2.y);
  s2.update(s1.x, s1.y);
  s1.display(s2.x, s2.y);
  s2.display(s1.x, s1.y);
  if (mousePressed == true) {
    s1.x = mouseX;
    s1.y = mouseY;
  }
}
```

50-20 cont.

The method used for calculating spring values in these examples is called the Euler (pronounced "oiler") integration technique. This is the easiest way to calculate these values, but its accuracy is limited. The Euler method works well for simple spring simulations, but it can cause problems with more complex simulations as small inaccuracies compound and cause the numbers to approach infinity. When this happens, people often say the simulation "exploded." For instance, shapes controlled by an Euler integrator might fly off the screen. A more stable but more complicated technique is the Runge-Kutta method. For sake of brevity, it is not covered here, but it can be found in other texts.

Exercises

1. *Move a shape using velocity and acceleration.*
2. *Make your own extension to the* `Particle` *class and use it in an example.*
3. *Devise a physical simulation using one of the classes derived from* `Spring2D`*.*

Synthesis 4: Structure and Interface

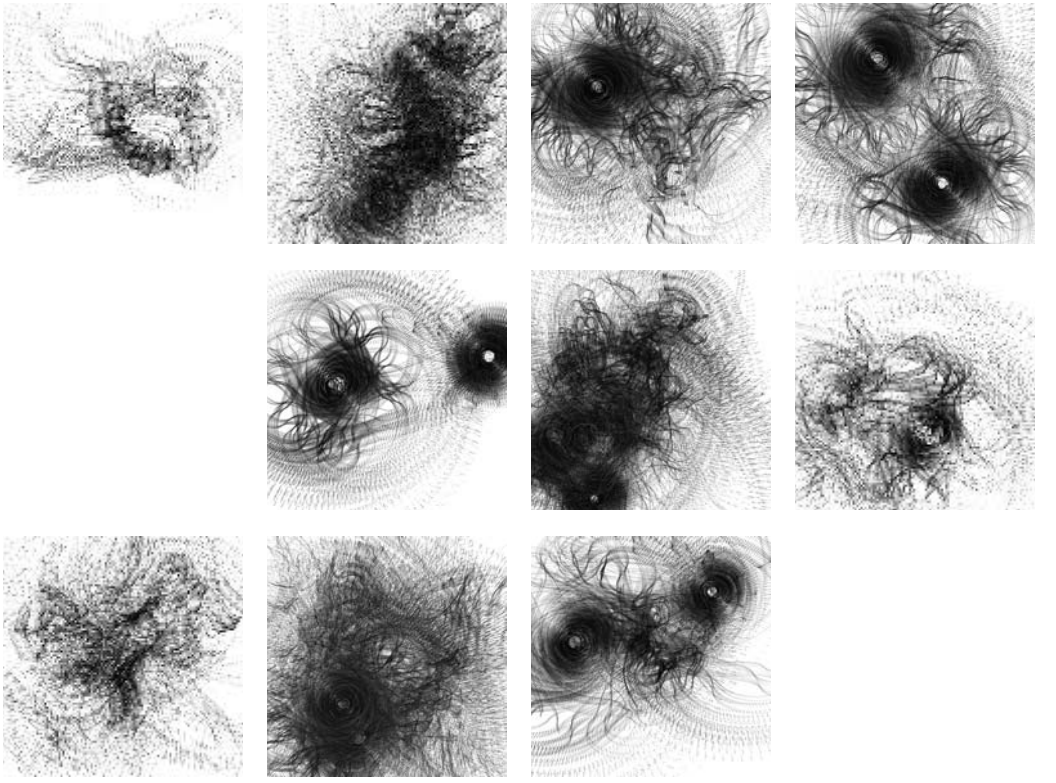
This unit presents examples that synthesize concepts from Structure 4 to Simulate 2.

The previous units introduced object-oriented programming, saving and importing files, creating graphical user interfaces, and simulating biology and physics. This unit focuses on integrating these concepts with an emphasis on object-oriented thinking. As mentioned, object-oriented programming is an alternative way to think about structuring programs. Knowing when to use object-oriented programming and how to structure the objects is an ability that develops with time and experience. The programs in this unit apply object-oriented thinking to previously introduced topics.

It's now possible to integrate elements of software including variables, control structures, arrays, and objects in tandem with visual elements, motion, and response to create exciting and inventive software. Because of space restrictions and our desire not to overwhelm the reader, this book omits discussion of many programming concepts, but the topics presented provide a solid foundation for diverse exploration.

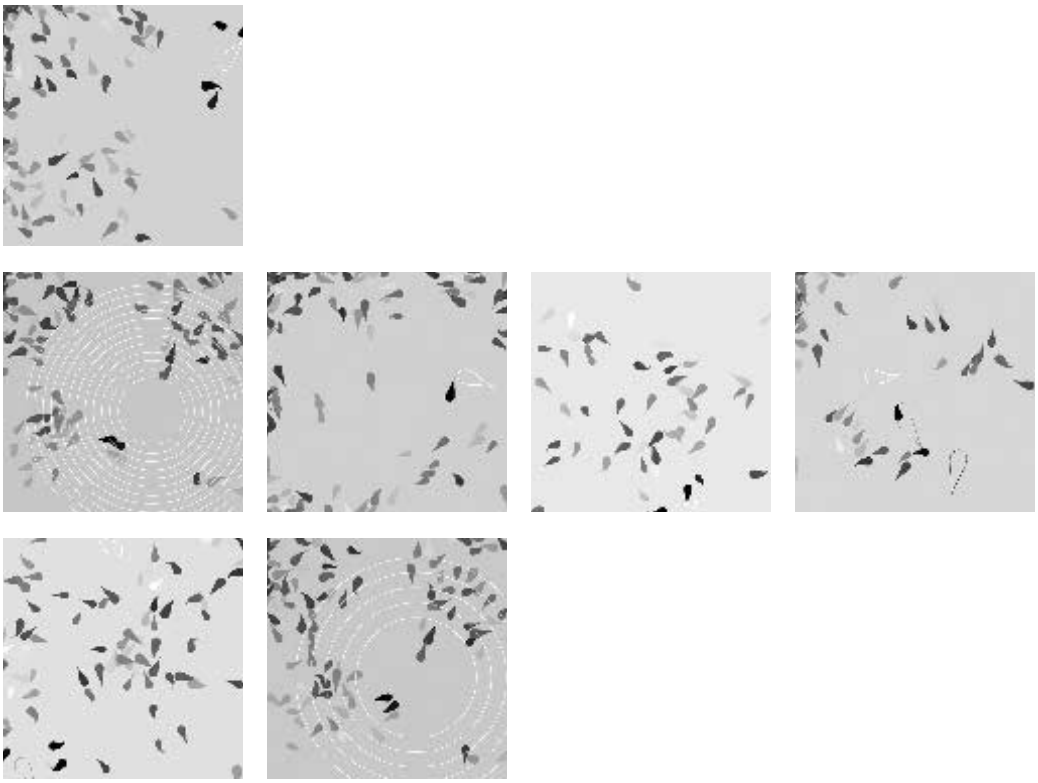
The programs presented in this unit are the most challenging in the book, but they include only ideas and code that have been previously introduced. They're challenging in their composition, but all of the components are built from the concepts discussed in this text. These programs include a game, drawing software, generative form, and simulations.

The four programs presented here were written by different programmers. Unlike most of the other examples in the book, which have been written in a similar style, each of these programs reflects the personal programming style of its author. Learning how to read programs written by other people is an important skill.
The software featured in this unit is longer than the brief examples that fill this book. It's not practical to print it on these pages, but the code is included in the Processing code download at www.processing.org/learning.



WithoutTitle. The images on this page were created with a sophisticated drawing program that combines elements of code from Motion 2 (p. 291), Structure 5 (p. 453), and Drawing 2 (p. 413). A dense thicket of lines circulates around the position of the cursor; moving the position of the cursor affects the epicenter and how the lines expand and contract.

Program written by Lia (http://lia.sil.at)

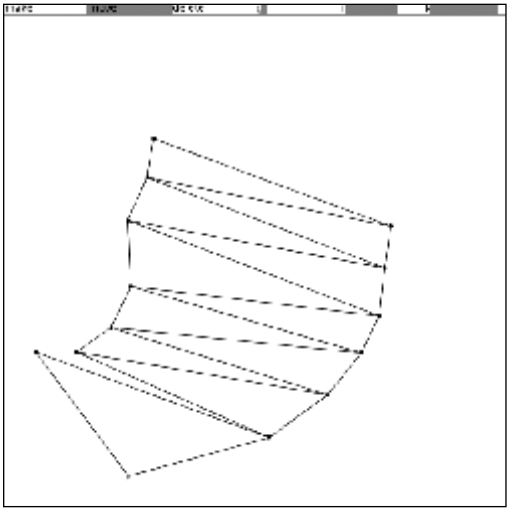


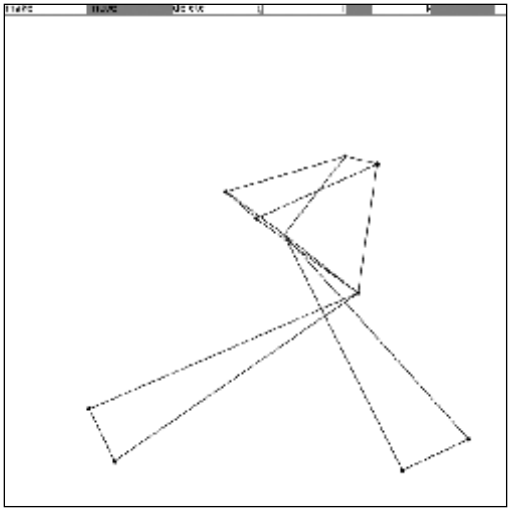
Pond. These images were generated from an implementation of Craig Reynolds's Boids rules, explained in Simulate 1 (p. 461). As each fish follows the rules, groups are formed and disperse. Clicking the mouse sends a wave through the environment and lures the creatures to the center. Each creature is an instance of the `Fish` class. The direction and speed of each fish is determined by the rules. The undulating tail is drawn with Bézier curves and moves from side to side in relation to the current direction and speed.

Program written by William Ngan (www.metaphorical.net)

Swingtree. This software simulates a tree swaying in the wind. Move the mouse left and right to change the direction and move it up and down to change the size. The connections between each branch are set by data stored in a text file. When the program starts, the file is read and parsed. The values are used to create instances of the `Branch` and `Segment` classes.

Program written by Andreas Schlegel (www.sojamo.de) at ART+COM (www.artcom.de)





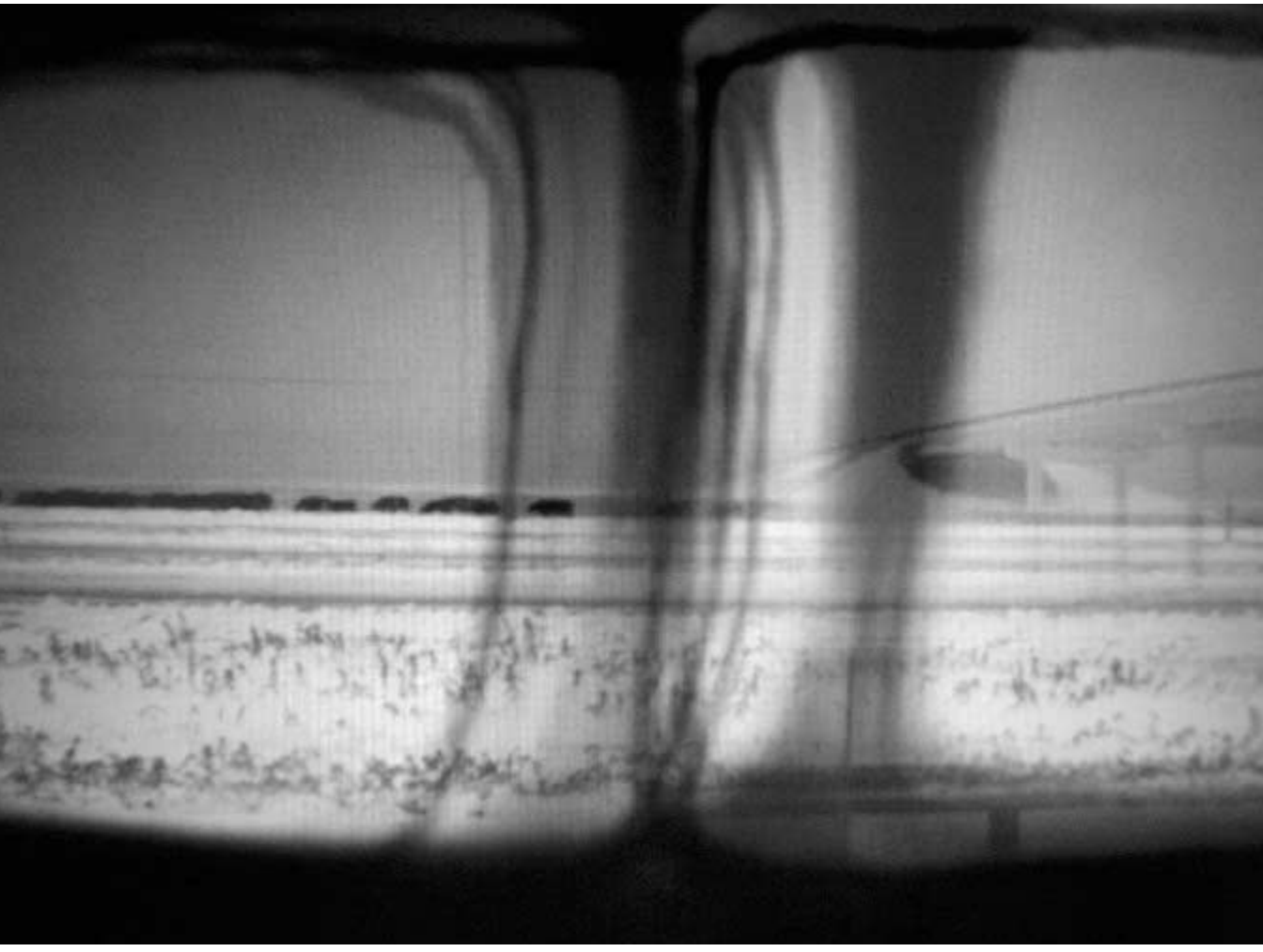
SodaProcessing. The Sodaconstructor (p. 263) connects simulated springs and masses to create fluidly kinetic creatures. This example is a simplified version of the Sodaconstructor, translated from Java to Processing. It builds on the ideas introduced in Simulate 2 (p. 477) and creates an interface from the ideas in Input 7 (p. 435). The GUI allows the user to create models by adding and deleting masses. Once you start a model, you can move each mass to see how the model reacts to force. The environmental gravity, friction, and stiffness can be changed by moving a slider left and right. This software integrates interface elements with spring and mass simulation.

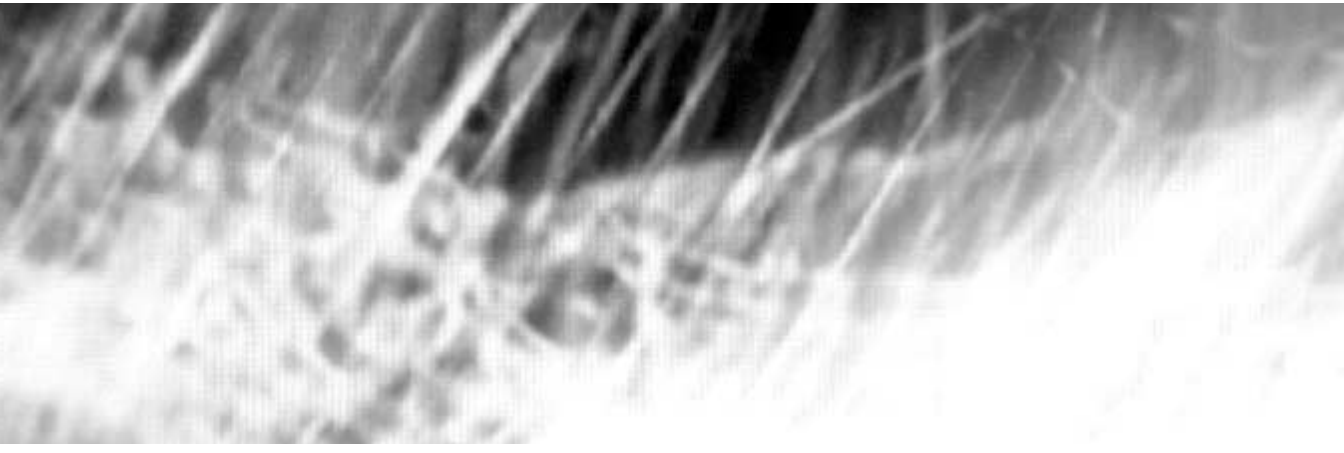
Program written by Ed Burton (www.soda.co.uk)

Still image from *Mini Movies*, 2005. Image courtesy of the artists.

Interviews 4: Performance, Installation

SUE.C. *Mini Movies*
Chris Csikszentmihályi. *DJ I, Robot Sound System*
Golan Levin, Zachary Lieberman. *Messa di Voce*
Marc Hansen. *Listening Post*





Mini Movies *(Interview with SUE.C)*

Creators	AGF+SUE.C
Year	2005
Medium	Performance, CD/DVD
Software	Max/MSP/Jitter, Radial
URL	www.minimoviemovement.com

What is *Mini Movies*?

Mini Movies *is a CD/DVD collaboration between the musician Antye Greie and the visual artist Sue Costabile. It is an audio visual collection of mini lives in an urban and political context. The liberation of the still image. A breakaway of recorded music.* Mini Movies *is also the current chapter in the live performance presented by AGF+SUE.C.*

Why did you create *Mini Movies*?

We began performing live sound and image together several years ago in an entirely improvisational fashion. Through the medium of live performance we discovered many commonalities between Antye's aural language and my visual language. Both of us see life as a series of miniature movies, some silent, some only a soundtrack waiting for the image to appear. The increasing consumability of the DVD format made it accessible to us as artists and we decided to present our own abstraction of the Hollywood feeling. The movie industry has made much of society accustomed to their mode of behavior and means of delivering entertainment. This is our way of slipping our own observations of life and audiovisual memories into the preestablished user interface and industry distribution network. In a live context our mini movies become larger than life, projected onto a giant screen and filling all available space with sound.

What software tools were used?

Our main studio production tools are Max/MSP/Jitter, Logic, Radial, and Final Cut Pro. As performers we both improvise a great deal, using very little recorded media and relying heavily on the human brain for interaction between sound and image. In our live performances we use Max/MSP/Jitter and Radial, along with a MPC, microphones, photographs, drawings, shiny objects, and many different miniature lighting rigs. These physical objects are an augmentation of the software and serve as a means through which we can interact more creatively with the tools.

Why did you use these tools?

Max/MSP/Jitter offers us the ability to create our own customized tools but also to easily incorporate elements of other people's tools. We were often meeting artists that were using this software environment to do wonderful and interesting things, and that led to each of us exploring and using it. We find the environment very flexible and open to experimentation. There is a structure to the language but there is also a freedom to explore sound and image as pure data. This leads fluidly to all kinds of manipulations, transformations, and translations. Our focus has been on developing a software-based environment that responds readily to human input and interaction. Antye uses her voice as the main input source and I use a live camera pointed at various layers of physical objects and lights which are animated by my hands.

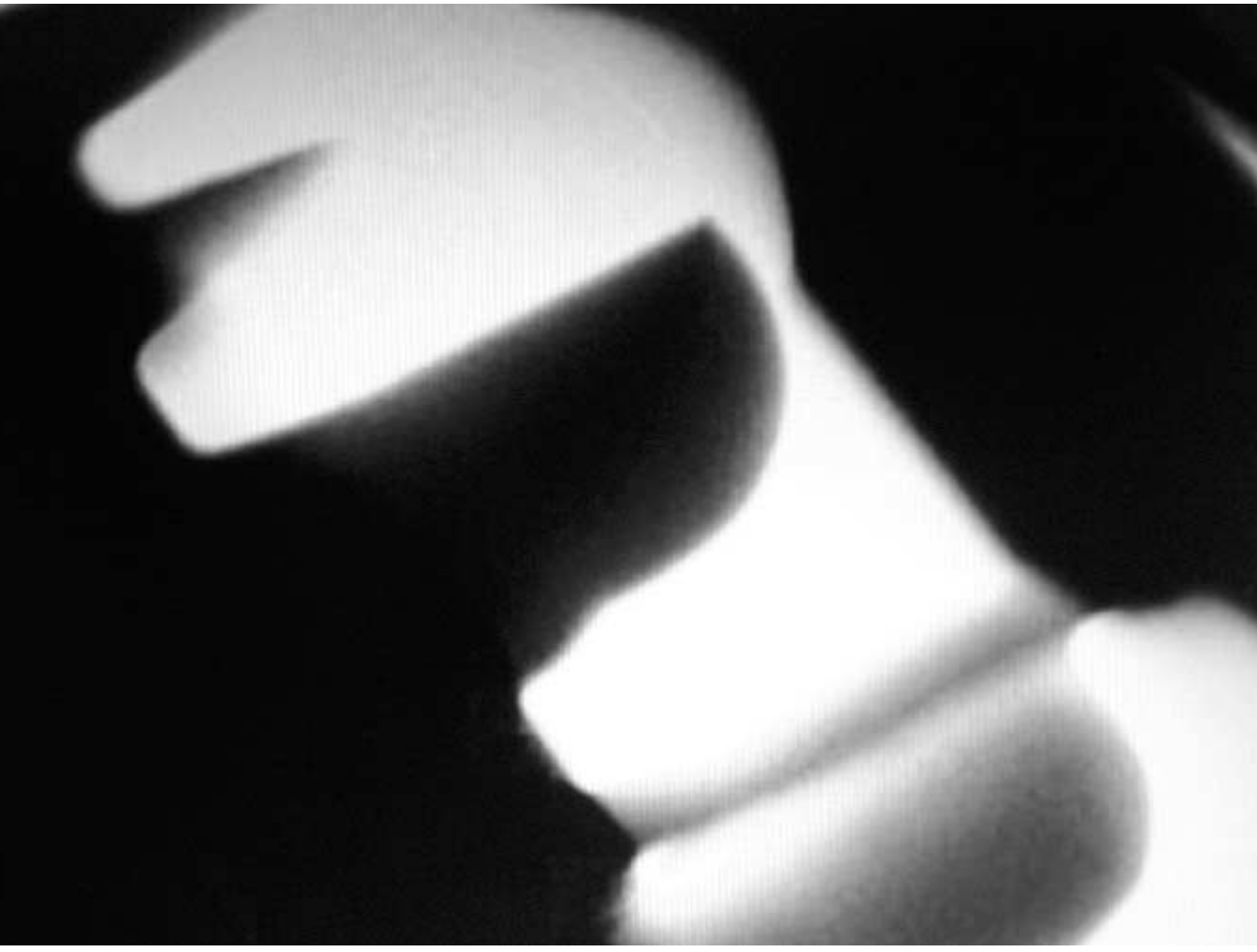
Still images from *Mini Movies*, 2005. Images courtesy of the artists.

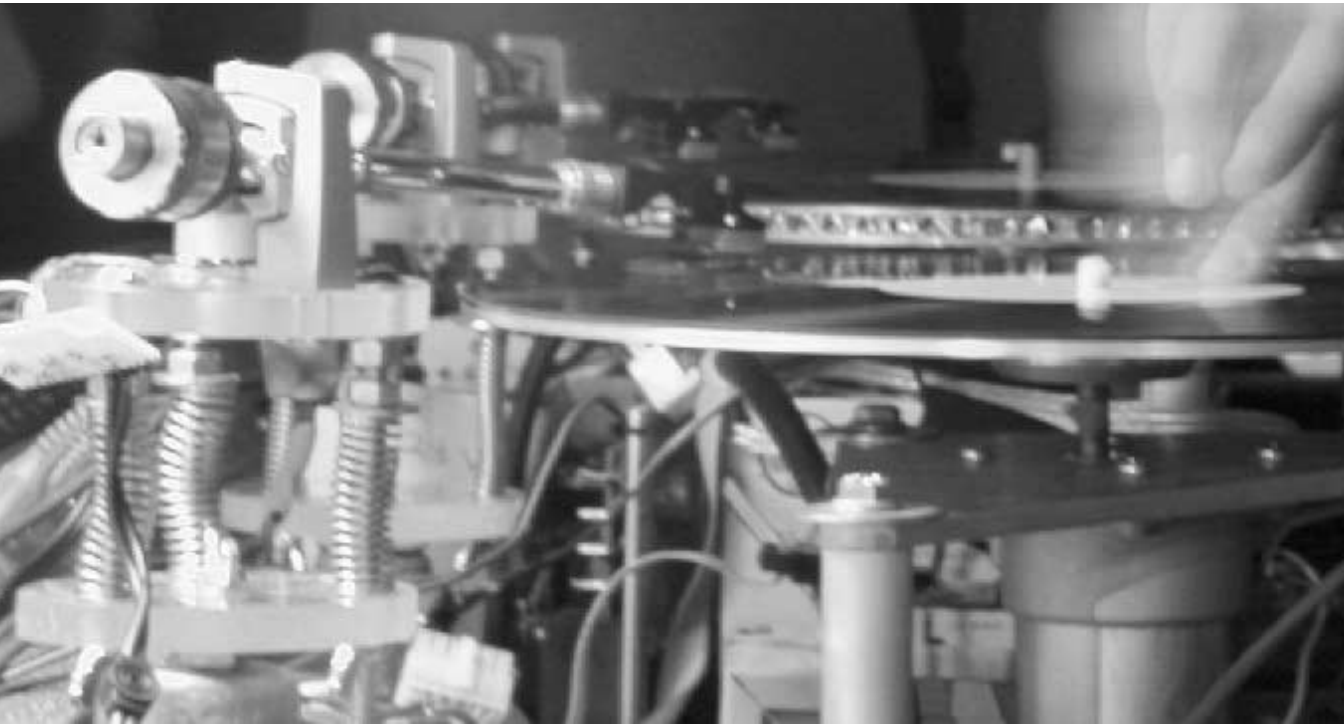
These analog inputs are processed by the software, which adds another layer of abstraction and emotion.

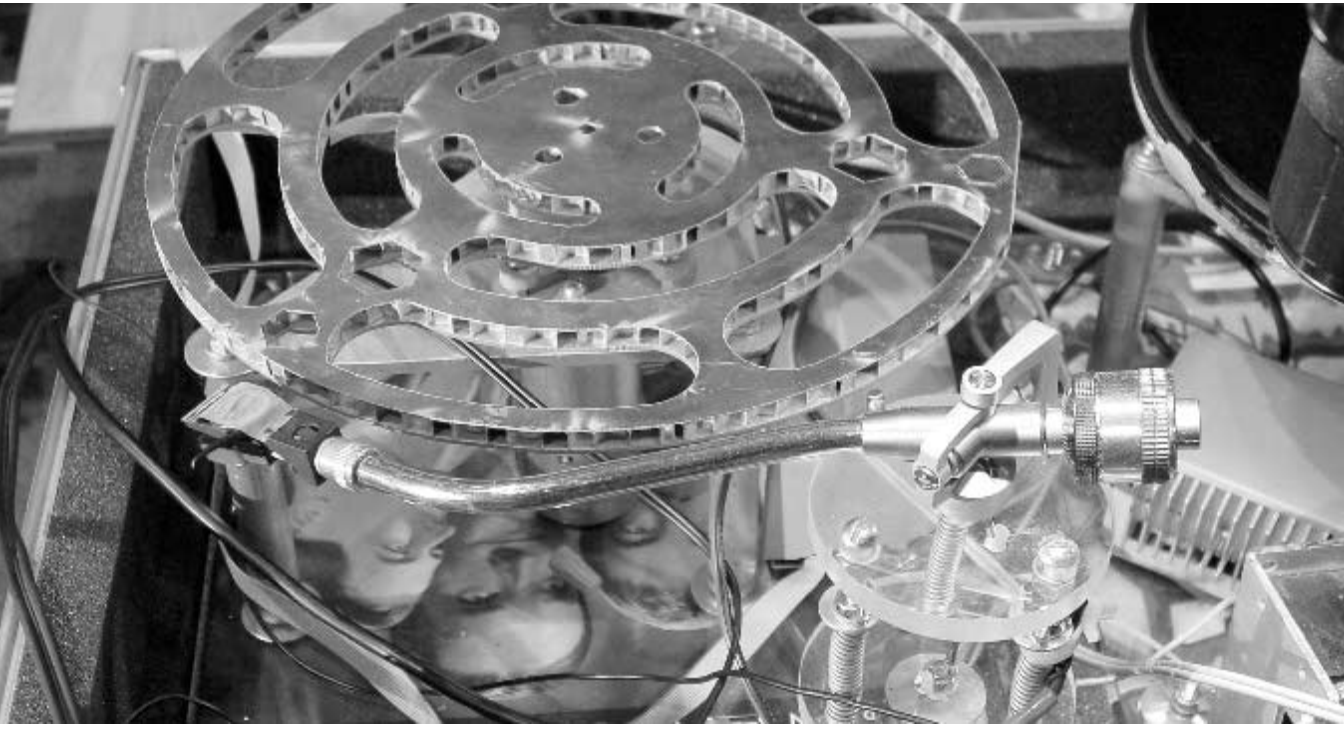
Why do you choose to work with software?

It wasn't an entirely conscious choice to work with software. We were both very intrigued by computers, the Internet, and digital media in general but once we discovered how expressive one could be with certain software products, we gravitated towards using them more and more. It provides us with broad artistic freedom since there is very little aesthetic preconception in the Max/MSP/Jitter environment. As artists we have been able to grow but not outgrow our tools. This is due in large part to the fact that we can program our own software tools inside of the environment. As our ideas change and expand the tools can do the same. As a video artist, I found the popular VJ software and hardware setups to be quite restricting. The idea for using a live camera as the only video signal input evolved out of frustration with working with a confined set of video clips and combining them in a limited number of ways. Jitter allows for a seemingly infinite array of compositing modes, and quite easily communicates with any digital camera. After a long period of experimentation, the software has become an instrument that I have learned to play. The program is a simple compositing machine but allows for complex interactions and animations, and the flexibility of the programming environment means that I can add and subtract features at will. Antye felt a similar restraint from popular live performance tools for audio until she discovered Radial. This software allows her to control samples of her voice in a very responsive way and leads to associations and multilayered narratives with an organic character. We both appreciate the unpredictability that our software brings to our live performance. It is an amplification of our live presence, which, as a performer, lies at the heart of the show.

Still images from *Mini Movies*, 2005. Images courtesy of the artists.







DJ I, Robot Sound System *(Interview with Chris Csikszentmihályi)*

Creators	Jonathan Girroir, Jeremi Sudol, Lucy Mendel, Galen Pickard, Andy Wong, and Chris Csikszentmihályi
Year	2001
Medium	Robot
Software	C++
URL	www.dj-i-robot.com

What is *DJ I, Robot Sound System*?

The project started as "Funky Functions" in 1998 out on the West Coast, but then we were Upstate and it was freezing cold so we had a lot of time, and we were reading our man Vonnegut with his joint Player Piano *and so it was all, like, unheimlich. So then we were looking at John Henry and automation and labor and we were going to call it "the DJ killer app" but then we sobered up because we're all into peace and the crew kept growing so we got with "DJ I, Robot Sound System" because of, you know, the prime directive. But the Vonnegut is about 1000 times better than any Asimov; he just didn't have any good hooks that we could bite. Get on it, Kurtis! You gotta do for us like you did for Michael!*

Why did you create *DJ I, Robot Sound System*?

[Laughing] Because DJs were all lazy! What with the picklz and the West Coast and the DMC, skills were getting mad, yo, but it wasn't really moving forward, just spinning faster. Then on the other side there were these clowns like Richie Hawtin who were selling product, trying to get rid of the vinyl. New devices for DJing that ignored the roots, that it started from a ghetto misuse of consumer electronics. I mean, when Matsushita heard what folks were up to they actually took the 1200 off the market! But now there were suits like Hawtin trying to replace the wax, all getting like "it's too heavy in the airports," or "it degrades," or "it takes too long to go digging when you've got Napster." (Yeah, that was back in the 00 when Napster was still from the block.) But all these new systems—like the CD mixers, "Final Scratch," and loads of mixing software—it was all basically saying vinyl's time had come.

Now that didn't make sense. There wasn't a single DJ I'd ever met who didn't love vinyl. Vinyl was there like—in the beginning?—we were supposed to play it like it was the word, just play it back, but then Selassie I told the systems in Jamaica "Cut it up, cheese!" and lo, they did cut. He was all "Lay your hands on it, son of God." It was like with the News, don't just sit back, read the truth, find the hidden word, and that's how it all happened. So why were these chump-ass marketers and engineers fronting on vinyl and saying it was time for being digital? It's like they were both "Love the caterpillar but do you have something more in a pirouette? and Love the tag but do you have it in a TIFF?"

So we were caught between fast and stupid, plus we hadn't been clubbing enough. So we got concentrated and stopped partying and bullshitting and started hacking and coding. It was raw, and shit. In the coffin it was all steel pipes and honeycomb aluminum composite, and old motors from a Xerox. But Technics represented with the tone arms, bless them, so we were coming strong. Explosions more than once [laughing], and lots of needles exchanged. On the laptop it was raw, right? Raw. Type conversions without casts, pointers to pointers to pointers,

DJ I, Robot Sound System, 2001. Images courtesy of the artists.

it was all raw. Raw! Plus, the software was only what got us to the door: there was this industrial networking protocol (rs485) and two microcontrollers per deck. That's what gave us the flexibility, see, to add another, and another, up to 128 decks for wreck and effects. We stopped at seven, though. That would have been too many stanky dollars, and we would've needed a semi. Plus it was already the threat to the DMC at just three decks, calling 'em out, saying, "What are you going to do about it?" Raw.

The very first time we had a human DJ in the lab—Bernard, a k a Flip 1—he was cool, no sweat, not nervous. He saw the future but he was all, "That's all it can do so far?" Cause he'd been at the DMC when he was eighteen but the robot was just a peewee, a few months old. And we were apologizing because one of our proportional integral derivatives was whack, K value way too high and we'd coded it in a constant (our bad), so it would go to some spot on the track and then pass it, then be like "whoa!" and go back but go too far. Sprung mass. So we were all apologies but Flip, he was like, "No, that's a feature. I've never heard that before." [Laughs] And he went back to his decks, and he practiced, and practiced, and it was a new sound. That's when we were all, "Gotta get up and be somebody!" We knew we weren't just faking the funk. I mean, the very first time a human heard the robot, he was changed.

Why did you write your own software tools?

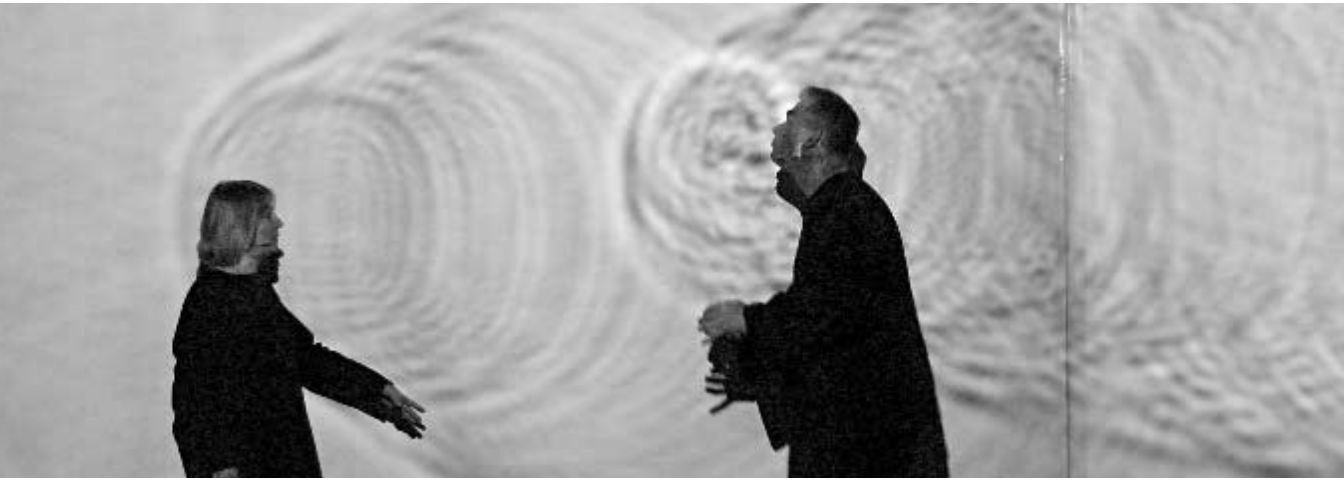
We peaked at a little under 10K of C++, though we could have gone all Rocky and toned it up, but it was flabby. Microsoft didn't help anyway. Never again. Anyone interested in machine control I'd tell her real-time Linux, period! Lot of versions, a little more code sprinkled here, there, for each performance. We had an FFT function to machine listen to a human DJ then play back their scratches right back at them. We had sequencers to record a performance and play it back. We had motion capture to parse a scratch, .0000015625 of a second accurate. It was raw. We had the gimmicks.

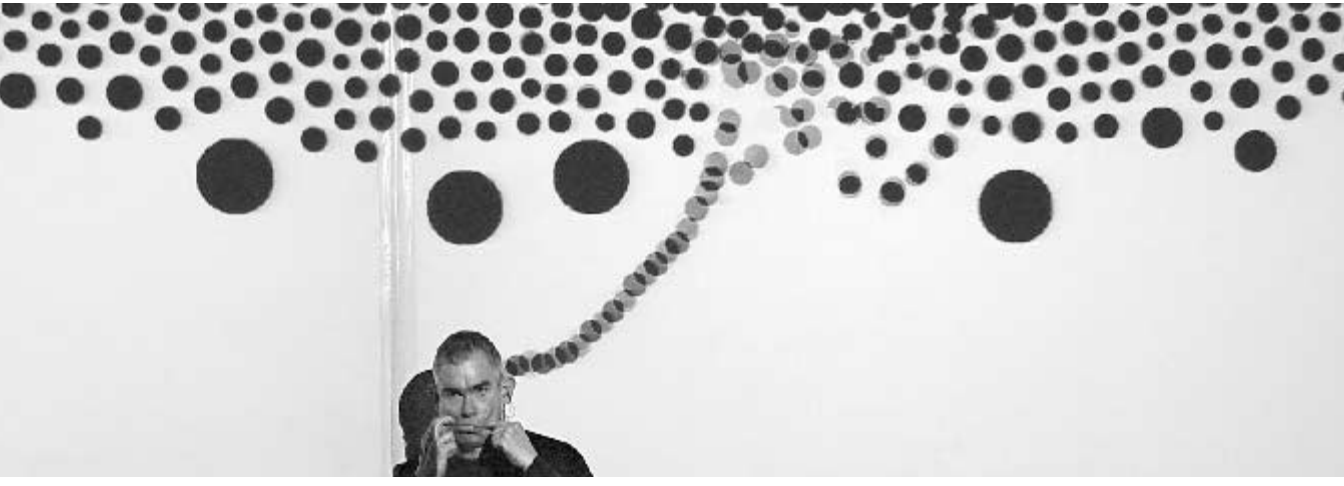
Why do you choose to work with software?

We worked with it all. Some mechanics, some software, some electronics, of course *the Jonzun Crew, and some sweat and tears. Software's not interesting. Beats are interesting. If software can help you find new beats, that's great. But there are a lot of ways to find new beats.*

DJ 1, Robot Sound System, 2001. Images courtesy of the artists.

DESTRUCTION
MIXES





Messa di Voce *(Interview with Golan Levin and Zachary Lieberman)*

Creators	Tmema (Golan Levin and Zachary Lieberman), with Joan La Barbara and Jaap Blonk
Year	2003
Medium	Interactive installation or performance with custom software
Software	Custom software for Windows, written in C++
URL	www.tmema.org/messa

What is *Messa di Voce*?

Messa di Voce *is an audiovisual performance and installation in which the speech, shouts, and songs produced by two vocalists are augmented in real time by custom interactive visualization software. The project touches on themes of abstract communication, synesthetic relationships, cartoon language, and writing and scoring systems, within the context of a sophisticated and playful virtual world.*

Our software transforms every vocal nuance into correspondingly complex, subtly differentiated and highly expressive graphics. These visuals not only depict the users' voices, but also serve as controls for their acoustic playback. While the voice-generated graphics thus become an instrument with which the users can perform, body-based manipulations of these graphics additionally replay the sounds of the users' voices, creating a cycle of interaction that fully integrates the visitors into an ambience consisting of sound, virtual objects, and real-time processing.

Messa di Voce *lies at an intersection of human and technological performance extremes, melding the unpredictable spontaneity of the unconstrained human voice with the latest in computer vision and speech analysis technologies. Utterly wordless, yet profoundly verbal,* Messa di Voce *is designed to provoke questions about the meaning and effects of speech sounds, speech acts, and the immersive environment of language.*

Why did you create *Messa di Voce*?

Messa di Voce *grew out of two prior interactive installations that we developed in 2002:* RE:MARK, *which explored the fiction that speech could cast visible shadows, and* The Hidden Worlds of Noise and Voice, *a multiperson augmented reality in which the users' speech appeared to emanate visually from their mouths. These installations analyzed a user's vocal signal and, in response, synthesized computer-graphic shapes that were tightly coupled to the user's vocal performance. After making these pieces, we had the feeling that we hadn't taken full advantage of everything we had learned about analyzing vocal signals. Although* RE:MARK *and* Hidden Worlds *were reasonably successful with a general audience, we wanted to step up to a much greater challenge: could we develop voice-interactive software that could somehow equal or keep pace with the expressivity of a professional voice artist?*

We invited the well-known experimental vocalist/composers Joan La Barbara and Jaap Blonk to join us in creating the Messa di Voce *performance. Although Joan and Jaap come from very different backgrounds—she works in contemporary art music, while he comes from a background in sound poetry—both of them share a practice in which they use their voices in extremely unusual and highly sophisticated ways, and both use a visual language to describe*

Joan La Barbara and Jaap Blonk perform *Messa di Voce*. Images courtesy of the artists.

the sounds they make. The software was really designed in collaboration with them—there are even specific sections or modules of the software that were directly inspired by improvisation sessions that we held together. Once the performance was finished, we realized that some sections could only ever be performed by trained experts like Joan and Jaap, but that other software modules could actually be experienced by anyone uninhibited enough to get up and yell or sing. We gathered up five or so of these—about a third of the original concert software—and that's how we redeveloped Messa di Voce *into an installation. We're proud that these software pieces could be used to good effect by expert vocalists, but even more pleased, in a way, that children can enjoy them too.*

What software tools were used?

We developed Messa di Voce *in C++, using the Metrowerks Codewarrior development environment. Some of the sound analysis was accomplished with Intel's commercial IPP library. We also incorporated a large number of open source sound and graphics toolkits, including OpenCV, OpenGL, and PortAudio.*

Why do you choose to work with software?

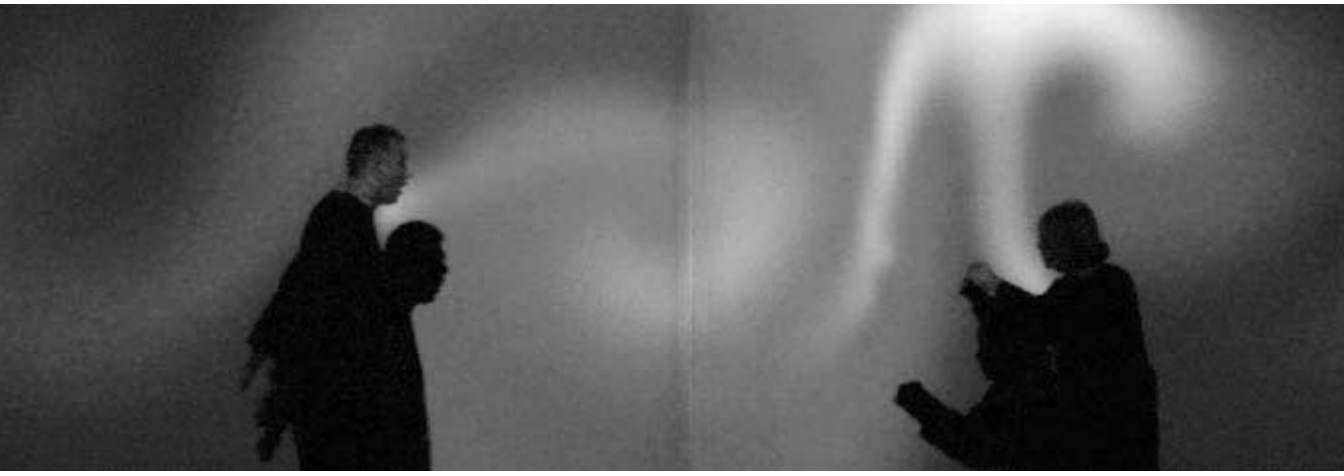
Because software is the only medium, as far as we know, that can respond in real time to input signals in ways that are continuous, linear or nonlinear as necessary, and—most importantly—conditional. The medium that we're interested in, to borrow a phrase from Myron Krueger, is response itself, and only software is able to respond in such a rich manner and with such a flexible repertoire.

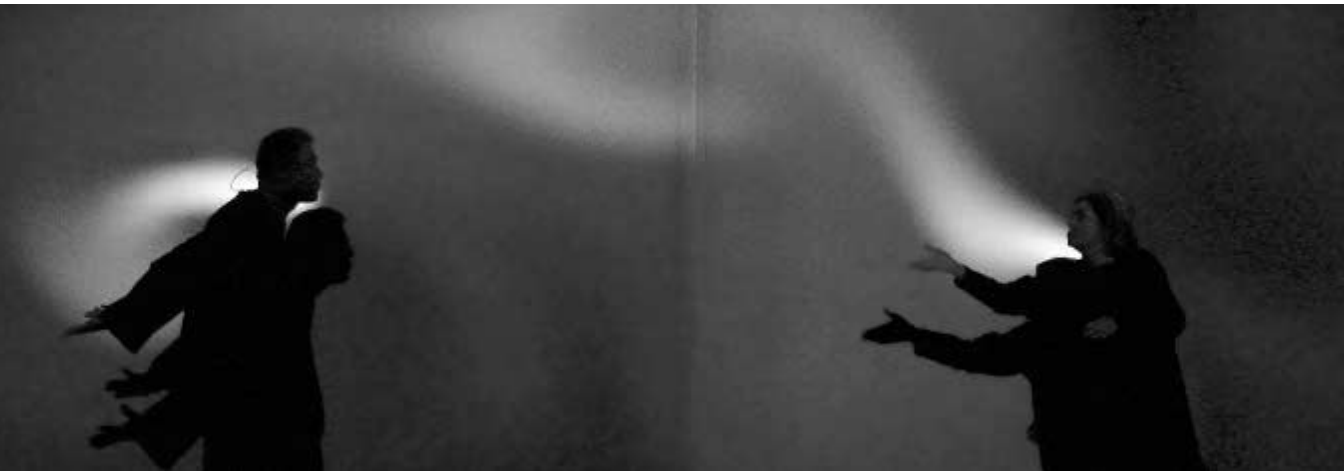
Why did you write your own software tools?

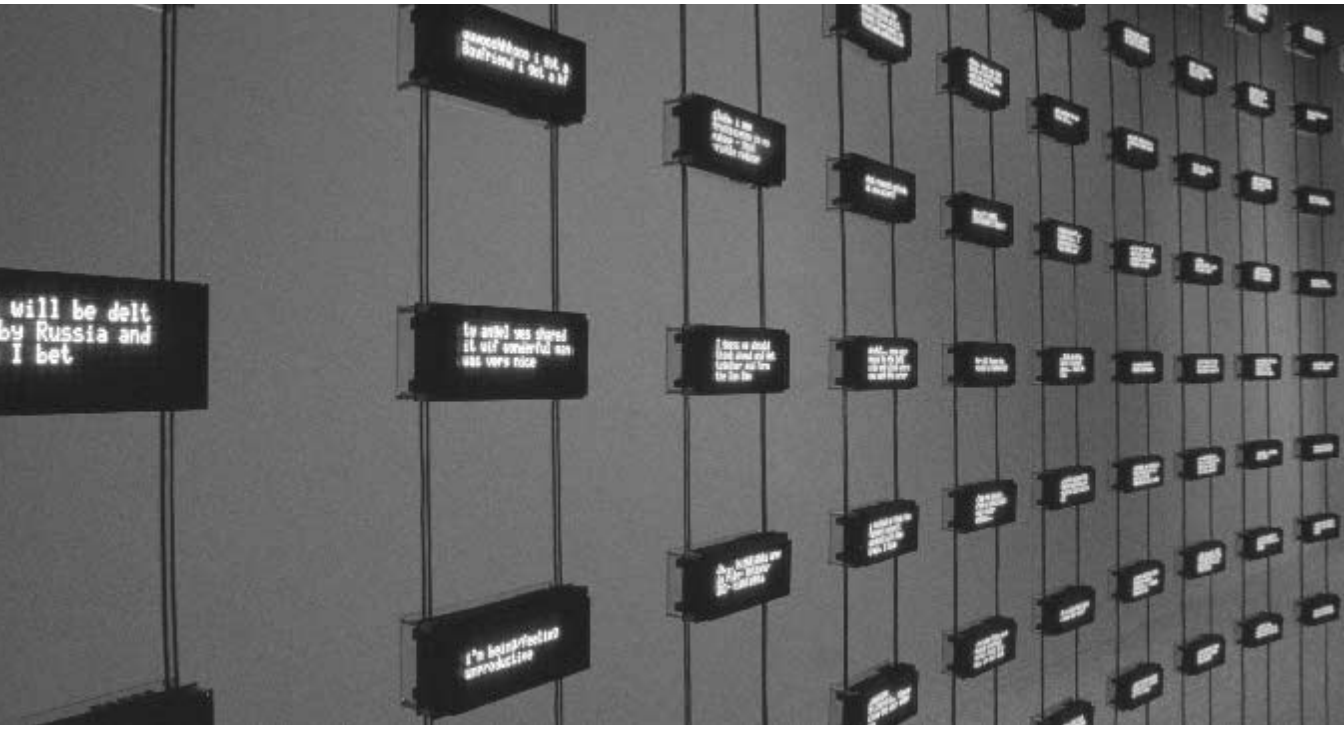
There isn't any other software that does what we want to do—and most importantly, that does it in the way we imagine it could be done. In the specific example of Messa di Voce*—although a significant aspect of the project is entirely conceptual (the idea of visualizing the voice in such a way that the graphics appear to emerge from the user's mouth), an equally important dimension is the quality and degree of craft that is applied to the creation of the work, and which is evident in its execution. Although the idea of* Messa di Voce *could have been implemented by any number of other artists (and indeed, systems illustrating related ideas have been created by others, such as Toshio Iwai, Josh Nimoy, Mark Coniglio, and Steven Blyth), we'd like to believe that nobody else could have created it with the particular character and texture we did.*

That said, it would be a mistake to believe that we wrote Messa di Voce completely from scratch. As we mentioned earlier, we made extensive use of both commercial and open source software libraries in order to develop it. It's not even clear what "completely from scratch" would mean for our project, unless we were to somehow construct our own CPU and develop our own assembly language for it! We incorporated features and functionality from the other software libraries whenever we didn't know how to do something ourselves, or could simply save time by doing so. Our work was built on the efforts of probably thousands of other people.

Joan La Barbara and Jaap Blonk perform *Messa di Voce*. Images courtesy of the artists.






will be delt
by Russia and
I bet

Listening Post *(Interview with Mark Hansen)*

Creators	*Mark Hansen and Ben Rubin*
Year	*2001–2002*
Medium	*Installation*
Software	*Perl, C, Max/MSP, C++, sh/tcsh, R*
URL	*www.earstudio.com/projects/listeningPost.html*

What is *Listening Post*?

Listening Post *is an art installation that culls text fragments in real time from unrestricted Internet chat rooms, bulletin boards, and other public forums. The texts are read (or sung) by a voice synthesizer, and simultaneously displayed across a suspended grid of 231 small electronic screens (11 rows and 21 columns).* Listening Post *cycles through a series of seven movements (or scenes) each with a different arrangement of visual, aural, and musical elements and each with its own data-processing logic.*

Why did you create *Listening Post*?

Ben and I met in November of 1999 at an event sponsored by Lucent Technologies (my former employer) and the Brooklyn Academy of Music. For our first project, we created a "sonification" of the browsing activity across a large, corporate website. Sonification refers to the use of sound to convey information about, or to uncover patterns in, data; it seemed like a reasonable place to start for a sound artist (Ben) and a statistician (me). We spent several weeks creating an algorithm that translated patterns of user requests into music. The mapping was pretty direct, differentiating traffic through major areas within the site (defined by a handful of top-level directories) and the depth to which people wandered (again, measured in terms of the site's directory structure). Unfortunately, it was tough to get anyone to take notice; even the site's content providers were hard-pressed to find a reason to listen. After a month or so we decided that perhaps navigation statistics (a by-product of the actions people take on the Web) were less interesting than the substance of their online transactions, the content being exchanged. We also agreed that the act of Web browsing wasn't very "expressive" in the sense that our only glimpse of the users came from patterns of clicks, lengths of visits, and the circle of pages they requested. These considerations led us to online forums like chat and bulletin boards. (Of course, this was early 2000; had we started our collaboration today, blogs or YouTube.com *or even* MySpace.com *might have been more natural next steps.)*

In retrospect, it was pretty easy to create a data stream from these forums, sampling posts from various places around the Web. Doing something with it, representing it in some way, responding to its natural rhythms or cycles, proved to be much harder. Text as a kind of data is difficult to describe (or model) mathematically. To make matters worse, online forums are notoriously bad in terms of spelling and grammar and many of the other bread-and-butter assumptions underlying techniques for statistical natural language processing. However, I think our interest in online forums went beyond summarizing or distilling their content (reducing the stream to a ticker of popular words or topics). Instead, we wanted to capture the moments of human connection; and in most cases these refused to be mathematized. Early in our process,

Listening Post, 2001-2002. Images courtesy of the artists.

we decided to let the data speak for itself in some sense, creating scenes that organized (or, formally, clustered) and subset the content in simple, legible ways.

Building on our experience with the Web sonification project, our first experiments with chat were sound pieces: A text-to-speech (TTS) engine gave the data a voice (or voices, as there might be up to four speaking at one time), and we created different data-driven compositional strategies to produce a supporting score. As we listened, however, we found ourselves constantly referring to a text display I hacked together to monitor the data collection. While we were led to this simple visual device to help make up for deficiencies in the TTS program ("Did someone really type that?"), it soon became an important creative component. This visual element evolved from a projection with four lines of text (at a live performance at the Kitchen in 2000), to a 10 by 11 suspended flat grid of VFDs, vacuum fluorescent displays (the first installation of Listening Post *at the Brooklyn Academy of Music in 2001), and finally to the arched grid of 231 VFDs (first exhibited in 2002 at the Whitney Museum of American Art).* Listening Post*'s visual expansion was accompanied by the introduction of a new TTS engine that let us literally fill the room with voices (as many as a hundred at one time).*

What software tools were used?

The behavior of each individual VFD is ultimately directed by an onboard microcontroller running a custom C program written primarily by Will Pickering at Parallel Development. The screens are then divided into 7 groups of 33 (each an 11 by 3 subset of the entire grid) and are fed messages by 7 servers that listen for commands to display text along columns or on particular screens. The basic screen server is written in Perl. One layer up, the arched VFD grid is choreographed via a series of scene programs, again written in Perl.

The audio portion of Listening Post *involves dynamic musical composition orchestrated by Max/MSP; messages are sent to Max from the scene programs via the Open Sound Control (OSC) protocol. It's worth noting that the programming interfaces for the audio and visual portions of* Listening Post *are very different; while Max is a visual programming environment, meaning that Ben directs output from one sound component to another by making connections in a "patch," I hack about in an Emacs window combining subroutines from a main scene module. The last major piece of software directly involved in the presentation of* Listening Post *is the TTS engine. Like Max, the TTS engine receives messages from the scene programs; unlike with Max, however, we had to write a custom C++ wrapper to handle the network communication. Aside from Max and the TTS engine, there are also other, perhaps less obvious, software components hidden in the system. The installation itself involves eight speakers and, during each scene, the voices and other musical elements move around the room. While Max handles much of this motion, a Yamaha Digital Mixing Engine (DME) is also used, which in turn requires a separate program for each of the scenes.*

Finally, we made use of a slightly different set of software tools during scene development. At a practical level, each new scene consists of a Perl program orchestrating the visual elements and controlling the overall scene structure and a Max patch/DME program pair creating the scene-specific audio. (At this point, we treat the VFD grid and the TTS engine as fixed-output devices whose programming does not change with scene; they respond to a predetermined set of commands.) The design of each scene emerged through an iterative process that cycled between making observations about the data and an evolving set of basic compositional scene elements. To make sense of our stream of text data, we relied on Perl for text parsing and feature

extraction, some flavor of UNIX shell for process control, and the R programming environment for data analysis, modeling, and statistical graphics.

Why did you write your own software tools?

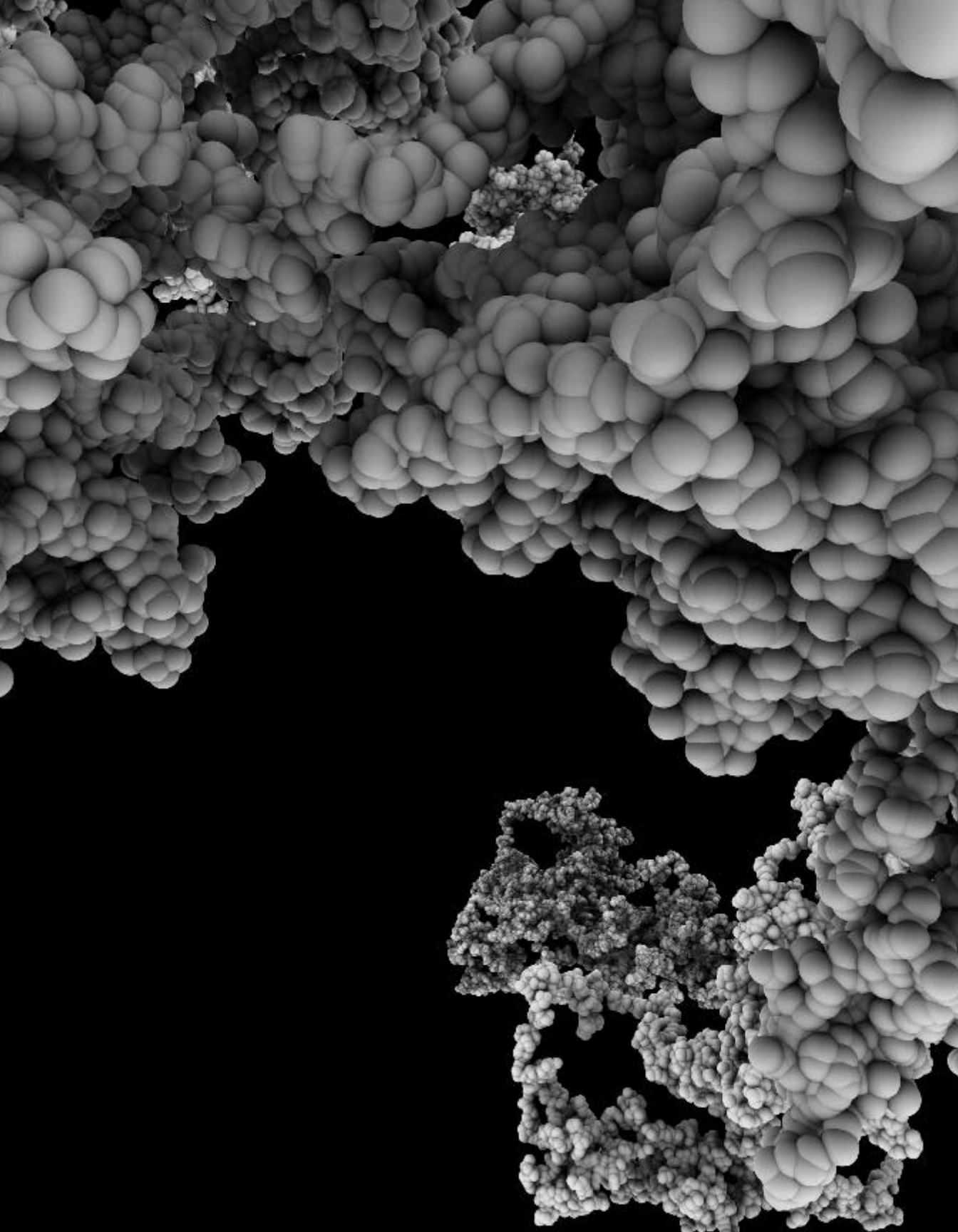
Given that the display "device" (the combined audio and visual components of the installation) was entirely new, we had little choice but to write our own software to control it.

For the most part, the software side of Listening Post *is constructed from what could be best described as "scripting languages." While it's a bit hard to pin down a precise definition for this term, it is often the case that such languages let you build up projects (programs or scripts) quickly in a fluid, interactive process that is distinct from programming in a "systems language" like C. For example, Perl is, by design, great for manipulating text (taking inspiration from previous UNIX shell facilities like awk); and over the years programmers and developers have created a stunning number of extensions to the language, including extensive tools for network programming. By working with Perl, I can output data to the VFD grid and quickly experiment with different scene dynamics, many of which involve parsing and computing with text. Using OSC, this same Perl program can also coordinate audio by sending messages to a Max/MSP process and to the TTS engine. Authoring scenes for* Listening Post *is an exercise in interprocess communication.*

Since 2000, the language Python has emerged as a strong competitor to Perl in this kind of application; Python even runs on many Nokia phones! If our development were taking place today, we would have to think seriously about programming in Python instead of Perl. The lesson here is that programming tools, and information technologies in general, are constantly in flux. If you choose software as a medium, your practice has to keep up with these changes. You need to be able to "read" a new language, assess its strengths and weaknesses, and determine which computations are "natural" (those that its designers have made easy to perform) and (if possible) why.

Why do you choose to work with software?

Software, or perhaps more generically computing, is the way I have come to know data.



Extension 1: Continuing...

It often takes a few years to become comfortable expressing ideas through software. The concepts are not difficult, but they represent a different way of thinking unfamiliar to most people. This book introduces elements of software within the context of the arts, with the aim of bringing ideas from computer programming within reach of a new audience. People have different aptitudes for learning computer programming, and the context in which it is introduced affects how well individuals learn it. The way into computer programming introduced in this book has proved effective for many people, but others interested in programming prefer a different path. The core software principles introduced in this text are applicable to many different programming languages and contexts.

This book is not about a specific programming language. It strives to make clear the abstract and obscure concepts behind programing, but to do this it's necessary to use examples from a language. The Processing Language and environment was chosen for this book because it was developed expressly for the purpose of teaching fundamentals of programming to the audience of designers and artists, and doing so in a way that fosters their future exploration of diverse programming contexts. You can explore programming further using Processing, and there are many other programming languages and environments to try. A programming language is usually designed for a specific context, and depending on the nature of your work, some languages will be more appropriate than others.

If this book has piqued your interest in programming, it is probably not the only book you'll want or need on the topic. While this book has discussed many of the ideas that are essential to writing software, it presents only the first steps. Everyone must decide for themselves how far they will proceed in learning more about the technical aspects of writing software. Some people will find the material covered in this book sufficient to realize their goals, and others will want to go further. One of the aims of this book is to enable the reader to benefit from more advanced and complete programming texts. There are many excellent books about programming, but the overwhelming majority of them assume some prior programming knowledge. This text covers the basics so as to make those more advanced texts accessible.

Extending Processing

The programming elements introduced and discussed in this book comprise a subset of the entire Processing language. This book covers all of the features found in the abridged reference, which is about half the Processing language. The complete reference offers

Karsten Schmidt. *Sunflow*, 2007. Software. Image courtesy of the artist.

more areas to explore; you can access it by selecting the "Reference" option from the Help menu or by visiting *www.processing.org/reference*. The complete reference includes functions for more advanced drawing techniques, 3D geometry, and data manipulation. The additional functions are demonstrated with examples in the reference and in the examples included with the Processing software.

By design, the Processing language has a narrow focus. It was built for creating images, motion, and responses to common input devices like the mouse and keyboard. Also by design, Processing can be extended beyond these areas. Processing libraries extend Processing to sound generation, networking, video input, and many other topics of media programming. Libraries are classified into two groups: core libraries and contributed libraries. The core libraries, including Video, Net, Serial, OpenGL, PDF Export, DXF Export, XML Import, and Candy SVG Import are distributed with the software and documented on the Processing website. Contributed libraries are created and documented by members of the Processing community. The contributed libraries range from physics simulations to computer vision to tools for facilitating data transfer. Newly contributed libraries are continually added, and it's hard to predict what will be developed in the future. A list of libraries is included with your software and is online at *www.processing.org/reference/libraries*. The OpenGL, Video, Net, Ess, and PDF Export libraries are explored in the following extension units.

A reference, or link, to a library must be added to a program before the library can be used. This link is one line of code that points to the location of the library's code. The line can be added by selecting the "Import Library" option from the Sketch menu, or it can be typed. For example, to use the PDF library, add this line to a program:

```
import processing.pdf.*;
```

This code tells the program to import all of the classes in the *processing.pdf* package. The asterisk (*) symbol is not used as the multiplication operator; it specifies that all the classes in the package should be imported.

The Processing libraries have been and will continue to be an exciting area of growth for the Processing environment. Rather than continual incorporation of new features within Processing, libraries will remain the primary way to extend the software. In a similar way a library is used to extend the core API in Processing, a tool can be used to extend the Processing Development Environment. Standard tools include a color selector and an autoformatter for code, but other developers have contributed tools that support features like formatting code to post to the Processing discussion board. Information about contributed libraries and tools can be found on the Processing development website: *http://dev.processing.org*.

The Processing Development Environment is intentionally minimal so that it is easy to use, but advanced users will find that it lacks some of the features included in many professional programming environments. Processing was designed for software sketches that consist of one to a dozen source files plus, maybe a library or two, and that draw to a display component. A larger project may become cumbersome to develop within the Processing Development Environment and can be loaded instead into a

different programming environment with more features. Eclipse (*www.eclipse.org*) is an open source development environment, primarily used for Java, that integrates well with Processing. Instructions on porting Processing projects to Eclipse can be found on the Processing site, and questions can be asked in the "Integration" section of *www.processing.org/discourse*. Most Java development environments should work, and the bridge between Processing and Java application development gets easier as members of the community contribute documentation and examples.

Beyond libraries and using other development environments with Processing, the software has also been extended into different domains through related but separate initiatives. Using the Processing Development Environment and similar programming languages, the Wiring and Arduino projects make it possible to program microcontrollers (the small computers found in electronic devices and toys), and the Mobile Processing project makes it possible to program mobile phones. The practice of programming is rapidly moving into these areas as computers become increasingly smaller and faster. The skills learned through using Processing can easily be transferred into these areas through Mobile Processing, Wiring, and Arduino. These projects are linked from the URLs *http://mobile.processing.org and http://hardware.processing.org*. They are introduced in more depth in Extension 7 (p. 617) and Extension 8 (p. 633).

We encourage you to actively participate in Processing. The software's success depends on the participation of community members. If you write a library for Processing, please consider sharing your code, in keeping with the way that Processing and its code are shared. If you write programs for your own enjoyment, as a part of your studies, or for professional practice, please upload them to the Web and share your discoveries. The community thrives when people share what they've learned and help answer questions for others.

Processing and Java

The Processing application is written in Java, a programming language introduced by Sun Microsystems in 1994. The language was originally designed for set-top boxes and was later adapted for the Web and named Java. In the years since, the focus of Java development broadened to include server-side applications, stand-alone desktop applications, and applications for smaller devices such as mobile phones.

When a Processing program is run, it is translated into Java and then run as a Java program. This relationship enables Processing programs to be run through the Web as Java applets or to run as applications for Linux, Macintosh, and Windows operating systems. It also allows Processing to make use of the extensive existing software components for Java.

Processing has a simplified programming style that allows users to program initially without understanding more advanced concepts like object-oriented programming, double-buffering, and threading, while still making those tools accessible for advanced users. These technical details must be specifically programmed in Java, but they are integrated into Processing, making its programs shorter and easier to read. While

Processing makes it possible to omit some elements of the Java language, it's also fine to leave them in. More information about the relationship between the two languages is shown in Appendix G (p. 686).

Other programming languages

If this book was your first introduction to computer programming, you're probably not aware of the many different language environments available for writing software. It's very common for a person to know how to program in a few different languages; new languages skills are often acquired, through knowledge of previously learned languages fades. Ben and Casey, for example, have written programs in languages including ActionScript, AutoLISP, BASIC, C, C++, DBN, Fortran, HyperTalk, JavaScript, Lingo, Logo, MATLAB, MAX, Pascal, PHP, Perl, Postscript, Python, and Scheme. This list may sound exotic and impressive, but it's not. After years of programming, one finds that different projects with different needs require diverse languages. In fact, many projects require a few languages to get the job done. For example, Ben's projects often use Perl to first manipulate text data, and then use Processing to display this data to the screen. There is no "best" language for programming, any more than a pencil is better than a pen or a brush; rather, it's important to use the tools that best suit your task. If you continue programming, you'll certainly learn a few different programming languages. Fortunately, after you have learned one language, learning others comes more easily.

A myriad of programming languages have been invented since people first started to program digital computers in the 1940s. In the 1950s, the first computer-generated images were created by scientists and engineers. These individuals were the only people with access to the scarce, expensive, and complex technology needed to make this work. Even in 1969, Jasia Reichardt, then a curator at the Institute of Contemporary Arts in London, wrote, "So far only three artists that I know have actually produced computer graphics, the rest to date having been made by scientists."[1] The works created during this time were typically made as collaborations between technicians at research labs and invited artists. The number of artists writing their own software has increased significantly in the last 35 years, especially since the introduction of the personal computer. Another increase in software literacy was engendered by the rapid adoption of the Internet in the mid-1990s.

Many programming languages have been appropriated by artists and designers to serve their own needs, and specialized languages have been written to fulfill the unique desires of this audience. The programming elements introduced in this book are relevant to many popular programming languages. The basic concepts of variables, arrays, objects, and control structures are shared with most other languages, but the ActionScript, C, C++, JavaScript, PHP, and Perl languages in particular share many specific syntax elements with Processing. ActionScript and JavaScript are the most similar because they are both based on a programming standard that is inspired by Java. The Java language was heavily influenced by C, and because C++, PHP, and Perl were all designed with references to C, they share similarities with Java and therefore with Processing.

This book and the Processing website contain information about additional programming languages. Appendix F (p. 679) introduces features of different programming languages and includes a brief description of selected languages commonly used within the arts. Appendix G (p. 686) compares Processing with Java, ActionScript and Lingo, two languages commonly used by artists and designers. Comparisons between Processing and Java, ActionScript, Lingo, Python, and Design by Numbers are published on the Processing website: *www.processing.org/reference/compare*.

Notes

1. Jasia Reichardt. "Computer Art," in *Cybernetic Serendipity*, edited by Jasia Reichardt (Praeger, 1969), p. 71.

Extension 2: 3D

Text by Simon Greenwold

For as long as people have represented the three-dimensional world on two-dimensional surfaces, they have invoked the help of machines. The 3D graphics we know today have their origin in the theory of linear perspective, developed less than 600 years ago by the Florentine architect Filippo Brunelleschi. He used a variety of optical devices to determine that all sets of parallel lines appear to the eye to converge at a single "vanishing point" on the horizon. Shortly after the technique was codified, artists such as Albrecht Dürer began devising machines to help produce convincing representations of 3D scenes on 2D picture planes. Anticipating modern methods such as ray-tracing, these devices are the ancestors of today's cheap graphics cards, which are capable of displaying more than a billion vertices per second on screen. Today, artists, designers, engineers, and architects all make use of computers to create, manipulate, and output 3D form.

A short history of 3D software

The earliest on-screen 3D graphics appeared in the 1950s, not on digital computers but using oscilloscopes, machines designed to trace voltages in electronic circuits. It took thirty more years for 3D graphics to enter the home by way of games for personal computers. A survey of the history of 3D graphics shows that the earliest adoption of many new technologies and practices came from gaming. A quick look at 3D graphics in popular games of the twentieth century is not a bad way to track the state of the art.

Among the earliest 3D games, Flight Simulator was released first in 1980 and survives as a Microsoft-owned franchise to this day. Early 3D graphics used the wireframe rendering technique to show all of the edges that make up a 3D form. This is the simplest method of rendering, but it results in a world that appears to be made entirely of pipe cleaners. Graphical adventure games like King's Quest (1983) advanced the discipline with detailed environments, occluding layers, and motion parallax—a perceptual depth cue whereby objects close to an observer move more quickly across the visual field than those far away. Games like Marble Madness, Crystal Castle, and Q*bert continued to draw on simple orthographic 3D representations without a great deal of innovation until John Carmack introduced Wolfenstein 3D in 1992, the original first-person shooter game. Since its introduction, this class of games has driven the consumer 3D market more than any other because of the tremendous computational demands involved in producing a convincing, real-time, immersive environment. At the same time that the first-person games began taxing real-time systems, games like Myst introduced richly rendered photorealistic imagery. The gameplay consisted solely in moving from one static image of a location to another, solving puzzles.

Michael Rees. *Putto8 2.2.2.2, 2003*, 2003. Fiberglass, paint. 78" × 62" × 72". Edition of 3. Image courtesy of the bitforms gallery , nyc.

As researchers, game companies, and artists strive to bring users a more completely immersive experience, they have moved graphics off the screen into a variety of architectural or wearable devices. Graphics departments in many universities now have "caves," giant inhabitable cubes with projected images covering every surface. Head-mounted displays, helmets, or glasses with screens directly in front of the eyes have been used by researchers since 1968. In Char Davies' 1995 artwork *Osmose*, an "immersant" wears a head-mounted display that allows her to navigate a real-time virtual environment consisting of twelve worlds simply by tilting her head and breathing. In contrast to such virtual reality systems, "augmented" reality suggests that rather than replacing an experienced reality with a virtual substitute, we can add to reality with virtual constructs. These systems often employ a handheld screen with 3D graphics overlaid onto a live video feed. In Simon Greenwold's *Installation* (2001), users can create virtual forms with a stylus and then "install" them into the real space of a room. The camera is attached to the back of the screen, resulting in an "eye-in-hand" experience in which the screen becomes a window into a world that contains a mix of real and virtual elements.

As full citizens of 3D space, computers are increasingly called upon to produce physical 3D artifacts. 3D printing technologies are currently a focus of research and product development. There are several common techniques for 3D printing, all of which are becoming faster, cheaper, and more widely available. One family of 3D printers, such as those from Z-Corp, works by depositing layer after layer of material (either itself molten or mixed with a fixative) and building up a form in a series of slices, a process called stereolithography. These techniques are used by architects for making models and by artists for producing sculpture. In *Putto8 2.2.2.2* (2003), the artist Michael Rees used 3D scanning, printing, and animation to produce grotesque, fanciful creatures formed through digital manipulation of scanned human body parts.

3D form

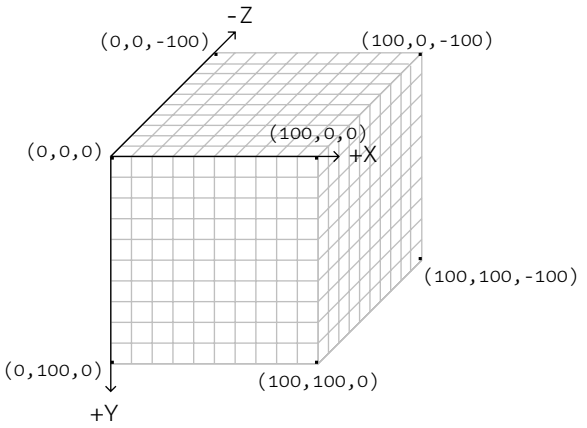
Form making is the first job in a typical 3D graphics workflow. This has traditionally been the responsibility of computer-aided design (CAD) software. The kinds of form that a piece of software helps users create and the kinds of manipulation it allows are tied directly to its internal representation of 3D form. There is active debate over which representation is best for each discipline. The only consensus is that the appropriate representation depends on the application. A mathematically exact representation of curved surfaces such as NURBS (Non-uniform Rational B-splines) makes a lot of sense for engineering applications because exact solutions are possible when the software does not need to approximate the curves. However, for 3D graphics, polygonal mesh representations allow for freer manipulation since the mesh need not be expressible as a pure mathematical formula. While a mesh can theoretically be made from any set of polygons, it is often convenient to work with meshes that consist entirely of triangles. Fortunately, since any polygon can be decomposed into a set of triangles, this does not

represent a geometric limitation. The process of turning some input form into a triangulated mesh is called triangulation.

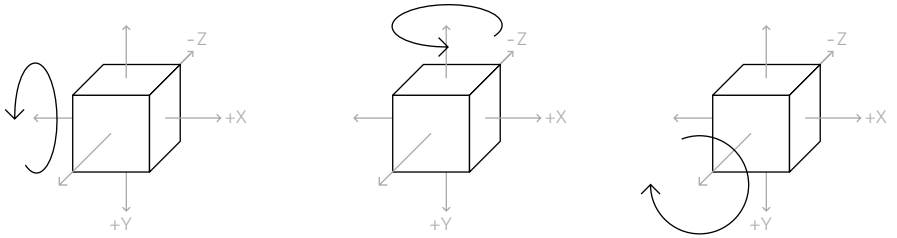
Both NURBS and mesh representations are surface representations, meaning that objects are defined exclusively in terms of their boundaries. This skin-deep representation is adequate for most 3D graphics applications, but may not be complete for engineering applications in which objects must be treated as true volumetric entities with properties such as density and center of mass. Common volumetric representations include Voxel, octree, constructive solid geometry (CSG), and binary space partition (BSP).

Most commercial 3D packages offer a library of simple forms such as boxes, cylinders, cones, and spheres. Each of these primitives has parameters to determine its shape and dimensions. A sphere has only one parameter to set its size, and a box has three, to set the width, height, and depth. Working with primitives is a bit like working with Lego blocks. A surprising amount of 3D work can be done simply by composing primitives. The level of detail possible is limited only by the scale of the building block relative to the scale of the composite.

These shapes are positioned into the 3D coordinate system. It builds on the 2D coordinate system of x-coordinates and y-coordinates and extends it with z-coordinates. Processing uses a coordinate system with the origin (0,0,0) in the front upper left with the z-coordinates decreasing as they move back from the front of the image:



Shapes are placed within the 3D coordinate system by definition of their coordinates and with the transformation functions. In Processing, the `point()`, `line()`, and `vertex()` functions have additional parameters to set coordinates in 3D, but other shapes must be positioned with transformations. The discussion of 2D transformations in Transform 2 (p. 137) applies to 3D with the addition of extra parameters. The `translate()` and `scale()` functions work the same way, with an added parameter for the z-dimension, but the `rotate()` function is replaced by three separate functions: `rotateX()`, `rotateY()`, and `rotateZ()`. The `rotateZ()` function is identical to the `rotate()` function, but `rotateX()` and `rotateY()` are unique to working in 3D. Each rotates the coordinates around the axis for which it is named:



Rotate around the x-axis Rotate around the v-axis Rotate around the z-axis

The `pushMatrix()` and `popMatrix()` functions also work identically in 3D. Pushing and popping the transformation matrix is particularly useful in 3D graphics to establish a place of operation and then restore an old one. Use the `pushMatrix()` function to push a transform onto the stack and set up the coordinate transform as you want it, including scaling, translations, and rotations. Create the local geometry, and then use `popMatrix()` to return to the previous coordinate system.

Before drawing 3D form in Processing, it's necessary to tell the software to draw with a 3D renderer. The default renderer in Processing draws only two-dimensional shapes, but there are additional options (`P3D` and `OPENGL`) to render 3D form. `P3D` is the simplest and most compatible renderer, and it requires no additional libraries. To use P3D, specify it as a third parameter to the `size()` function. For example:

```
size(600, 600, P3D);
```

The `OPENGL` renderer allows a sketch to make use of the OpenGL library, which is designed for high-performance graphics, particularly when an accelerated graphics card, such as those used for gaming, is installed on the computer. This makes it possible for programs to run more quickly than `P3D` does when lots of geometry or a large display size is used. Programs utilizing the `OPENGL` renderer can also be viewed online, but the download may take longer and may require a newer version of Java to be installed on the user's computer. To use the `OPENGL` renderer, select "Import Library" from the Sketch menu to add this line to the top of the program:

```
import processing.opengl.*;
```

and then change the `size()` function to read

```
size(600, 600, OPENGL);
```

After a 3D renderer is selected, it's possible to start drawing in 3D.

Example 1, 2: Drawing in 3D (p. 539)
When an object moves or rotates in 2D, its shape does not change. A 3D shape, on the other hand, grows larger or appears to spin away from the viewer as it is rotated and

moved because the three-dimensional space is drawn with a simulated perspective. Example 1 demonstrates rotating a rectangle around the x- and y-axis. As the mouse movement continuously changes the rotation values, the form appears as a square, a line, and a range of parallelograms. Example 2 changes the position of a sphere, box, and a word with `mouseX`, `mouseY`, and `translate()`. As their position changes, the rectangle and word appear differently, but the sphere looks the same. Pressing a mouse button runs the `lights()` function to illuminate the scene and shade the volumes. The sides of a shape are each at a different angle in relation to the lights and reflect them differently.

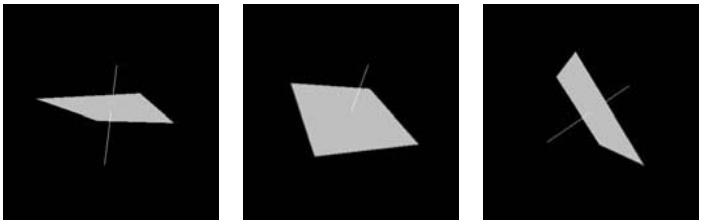
Example 3: Constructing 3D form (p. 540)
3D form is created with vertex points similarly to the way 2D shapes were created in Shape 2 (p. 69), but the extra z-coordinate makes it possible to define volumetric surfaces. This example demonstrates a function for generating a parameterized cylindrical shape and controls its orientation with the mouse. The `drawCylinder()` function has four parameters to set the top and bottom radius, the height, and the number of sizes. When the parameters are changed, the function can create different forms including a pyramid, cone, or cylinder of variable resolutions and sizes. The `beginShape()` function is used with values from `sin()` and `cos()` to construct these extruded circular forms.

After the form-making process is complete, if a user wishes to save the generated form a file format must be chosen. Every commercial package has its own preferred file format, some of which have become de facto industry standards and each of which has pros and cons. Many of the 3D file formats are proprietary, and information about them comes purely from reverse-engineering their contents. Two frequently used formats are DXF and OBJ, and each is used for different reasons.

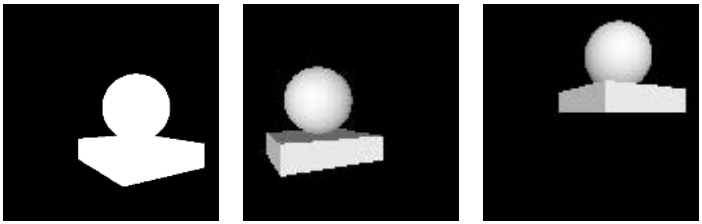
DXF is one of the native formats of AutoCAD and is a hassle to read or write. It is useful only because practically everything supports it, AutoCAD being a dominant standard. DXF is a poorly structured and enormous format. It has been around since 1982, becoming more complicated with every release of AutoCAD. There is a small set of DXF that is only moderately onerous to write out, so it is possible to use DXF as an export format. However, because other software may write files that use parts of the DXF specification that are hard to interpret, it is not useful as a general method of importing shape data.

OBJ, developed initially by Silicon Graphics, is useful for exactly the opposite reasons. Unlike DXF, it is not supported everywhere, but it is a dream to read and write. OBJ also has some sections that are not totally straightforward, but it is easy to configure exporters not to write that kind of data, so it becomes useful as both an import and export format.

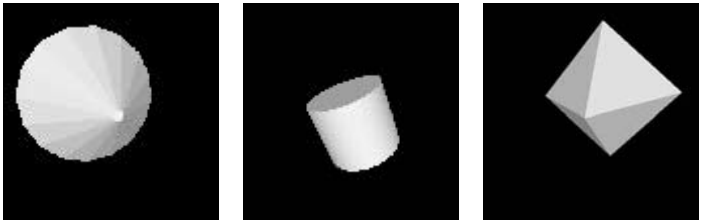
Example 4, 5: DXF export and OBJ import (p. 541, 452)
For these examples, the Processing DXF library is used to export a DXF file and the OBJ Loader library, written by Tatsuya Saito, is used to load an OBJ model. The DXF library is used to write triangle-based graphics (polygons, boxes, spheres, etc.) to a file. The OBJ



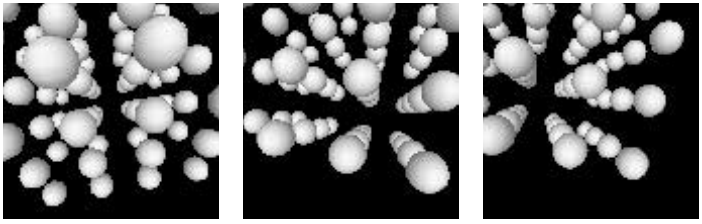
Example 1
The mouseX and mouseY values determine the rotation around the x-axis and y-axis.



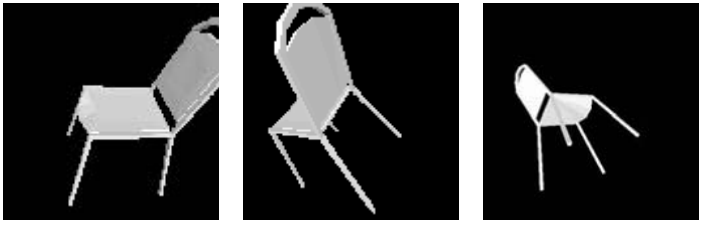
Example 2
Draw a box and sphere. The objects move with the cursor. A mouse click turns the lights on.



Example 3
Shapes are constructed from triangles. The parameters for this shape can transform it into a cylinder, cone, pyramid, and many shapes in between.



Example 4
The geometry on screen is exported as a DXF file.

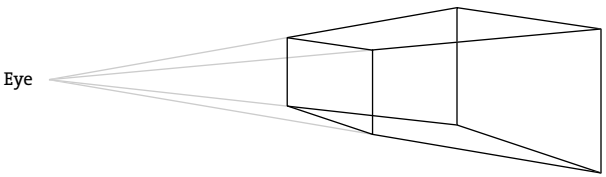


Example 5
Load a pre-constructed OBJ file and the mouse moves it from left to right.

Loader library can load the coordinates and material data from an OBJ file and then render this data in different ways. The DXF library is included with Processing, but the OBJ Loader library is linked from *www.processing.org/reference/libraries*. Each library must be added to a program before it is used. This is explained on page 520. Example 4 uses the `beginRaw()` function tell the program to start recording geometry and `endRaw()` finish the file. It is saved into the file *output.dxf*, which is saved in the current program's folder. Example 5 loads a simple `OBJ` object and the mouse is used to change its rotation. The `load()` method reads the model into an `OBJModel` object, and the `draw()` method displays it to the screen.

Camera

All renderings rely on a model of a scene and a camera (eye) that observes it. Processing offers an explicit mapping of the camera analogy in its API, which is derived from OpenGL. The OpenGL documentation (available online; search for "OpenGL Red Book") offers an excellent explanation of the workings of its camera model. The perspective camera as modeled by OpenGL and Processing can be defined with just a few parameters: focal length, and near and far clip planes. The camera contains the "picture plane," the theoretical membrane at which the image is captured. In a real camera, the film (or digital sensor) forms the picture plane. The focal length is a property that determines the field of view of a camera. It represents the distance behind the picture plane at which all the light coming into the camera converges. The longer the focal length, the tighter the field of view—it is just like zooming in with a telephoto lens.



Perspective viewing volume

Rendering requires three transformations. The first transformation is called the *view* transformation. This transformation positions and orients the camera in the world. Establishing a view transformation (expressed as a 4 × 4 matrix) implicitly defines "camera space," in which the focal point is the origin (the upper-left corner of the display window), the positive z-axis points out of the screen, the y-axis points straight down, and the x-axis points to the right. In Processing, the easiest way to establish a view transformation is with the `camera()` function. On top of the view transformation is the *model* transformation. Generally these two transformations are multiplied into each other and considered to be a unit known as the *model-view* matrix. The model transformation positions the scene relative to the camera. Finally there is the *projection* transformation, which is based on the camera's internal characteristics, such as focal length. The projection matrix is the matrix that actually maps 3D into 2D.

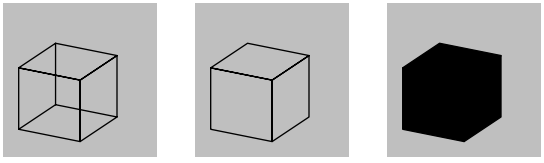
Processing by default establishes a set of transformations (stored internally as `PMatrix` objects called `projection` and `modelview`) that make the picture plane in 3D coincide with the default 2D coordinate system. Essentially it is possible to forget entirely that you are in 3D and draw (keeping z-coordinate equal to zero) as though it were a 2D canvas. This is useful because of the integration of 2D and 3D in Processing, although it differs from the default of other 3D environments. It also means that the model's origin is translated significantly in front of the picture plane so that it can often be viewed without further transformation.

Materials and lights

After 3D form is constructed and transformed, it is typically rendered into static images or animations. The state of the art advances so quickly that even graphics produced three years ago look crude today. The primary goal of software rendering has been photorealism—the images produced should be indistinguishable from photographs. Recently, however, there have been significant innovations in nonphotorealistic rendering, which attempts to produce stylized images. Cartoon, charcoal, or painterly renderers attempt to mimic the effects of a human hand and natural materials. Cartoon rendering, in which the edges of objects are identified and heavily outlined, is now used in some in real-time 3D games.

The work of 3D rendering is primarily the mathematical modeling and efficient computation of the interaction of light and surface. Ray-tracing and more advanced variants are the basis of most popular methods of rendering. Ray-tracing models rays of light emerging from a light source and bouncing around the surfaces of a scene until they hit the picture plane. This is computationally costly and fails to predict certain important phenomena of natural lighting such as the "color bleed" when one colored surface reflects onto another. Techniques like radiosity model "global illumination," which accounts not only for light that comes directly from predefined light sources but also light reflected off of the regular surfaces in a scene.

There are three methods of rendering that do not require calculating lighting: wireframe, hidden-line, and flat-shaded:



Wireframe Hidden line Flat shading

Wireframe is the simplest rendering model. It renders lines and the edges of polygons in their basic color. This is achieved in Processing by drawing with a stroke color and without a fill. Next in complexity is hidden-line. In this model only edges are drawn, but they are not visible where they would be occluded by solid faces. Processing does not support this directly, but it is easy to simulate by using a fill identical to the background

color. The last unlit model is flat-shaded, in which faces of objects are colored, but only using their base fill color.

Lighting and surface materials must be modeled for images to look more realistic. The techniques used for calculating real-time lighting are different from the ray-tracing and radiosity methods discussed above. Those are far too computationally expensive for fast rendering, although it is a safe bet that the processing power available in the future will be able to supply it. Instead, several common simplified lighting techniques are used for real-time graphics. In order to understand them, we need to introduce the model of light and surface-material interaction that nearly all real-time 3D uses.

The first type of light interaction with a surface has no direction and is called *ambient*. It is meant to model light in the environment that has bounced around so much it is impossible to know where it originally came from. All natural daytime scenes have a considerable amount of ambient light. Ambient lights are specified with the `ambientLight()` function, and they interact with the ambient color of a shape. The ambient color of a shape is specified with the `ambient()` function, which takes the same parameters as `fill()` and `stroke()`. A material with an ambient color of white (255, 255, 255) will reflect all of the ambient light that comes into it. A face with an ambient color of dark green (0, 128, 0) will reflect half of the green light it receives but none of the red or blue.

Shapes are treated as a set of *faces*. For example, each of the six sides of a cube is a single face. Each face has a *normal*, a direction vector that sticks straight out of it, like an arrow that extends perpendicularly from the center of the face. The normal is used to calculate the angle of a light relative to the object, so that objects facing the light are brighter and objects at an angle are less so. Ambient light, since it is without direction, is not affected by a surface's normal, but all other types of light are. The material reflects light in two ways. First is diffuse reflection. A materials has a diffuse color that affects the amount of light that scatters in all directions when it is hit by light. When light hits the surface head-on (in line with the normal), the surface reflects all of its diffuse color; when the light is at 90 degrees to the normal, the surface reflects none. The closer to the normal a light hits a surface, the more diffuse light it will reflect (this is calculated using the cosine of the angle between the incoming light and the normal). The more diffuse a surface is, the rougher and less shiny it appears.

Often the ambient and diffuse components of a material are manipulated together. Physically, they are essentially the same quantity. The `fill()` function in Processing sets both together, but the ambient color can be controlled separately with the `ambient()` function.

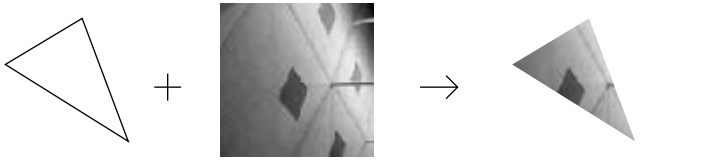
The second directional component of light is specular reflection. This is light that is bounced off of the surface reflected across the normal. The more specular reflection a material has, the more reflective it appears. A perfect mirror, for example, has no diffuse reflection and all specular reflection. Another parameter called *shininess* also factors into specular reflection. Shininess is the rate of decay of specular reflection as the incoming ray deviates further from the normal. A high shininess will produce very intense bright spots on materials, as on shiny metal. Lower shininess will still allow for specular reflection, but the highlights will be softer.

The last component in surface lighting is *emissive* color. This is color that is not tied to any incoming light source. It is the color with which a surface glows on its own. Since emissive faces are not themselves light sources, they do not glow in a very realistic way, so emissive color is not often useful. Mostly they are used in very dark scenes when something must show up brightly, like a headlight.

There are several different types of light that can be added to a scene: directional, ambient, point, and spot. Directional lights are the only kind that do not have a position within the scene. These lights closely approximate a light source located infinitely far from the scene. They hit the scene at a specific direction irrespective of location, so they are a good way to simulate sunlight. Other light types have positions and are therefore subject to *falloff*, the diminishing of light intensity with distance. In the real world, light intensity falls off proportionally to the square of the distance to the source. In 3D scenes, it is common to use little or no falloff so that fewer light sources are needed, which is more computationally efficient. Also, the extra light is needed because light doesn't bounce around a simulated scene the way it does in real life.

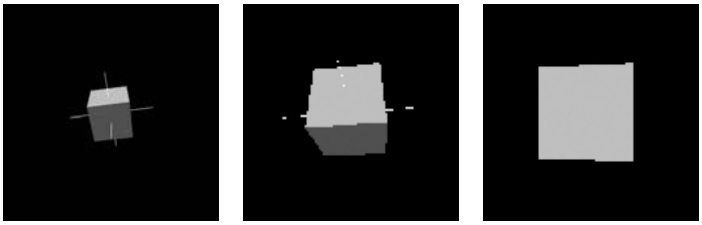
The simplest positioned lights are ambient lights. These are wholly nondirectional, and their position is used only to determine their range (falloff). Point lights model a bare bulb hanging in a room. They have a position, and their directionality radiates outward from that position. They shine equally in all directions—but only in a specific direction relative to any other point in the scene. Spot lights have the most parameters: position, direction, falloff, angle, and concentration. The angle affects how wide the spot light is open. A tiny angle casts a very narrow cone of light, while a wider one lights more of the scene. The concentration parameter affects how the light falls off near the edge of the cone angle. Light in the center is brighter, and the edges of the cone are darker. Spot lights require more calculations than other types of lights and can therefore slow a program down.

The texture of materials is an important component in a 3D scene's realism. Processing allows images to be mapped as textures onto the faces of objects. The textures deform as the objects deform, stretching the images with them. In order for a face to have an image mapped to it, the vertices of the face need to be given 2D texture coordinates:



Geometry Texture Texture mapped to geometry

These coordinates tell the 3D graphics system how to stretch the images to fit the faces. Good texture mapping is an art. Most 3D file formats support the saving of texture coordinates with object geometry. Textures are mapped to geometry using a version of the the `vertex()` function with two additional parameters, *u* and *v*. These two values are the x-coordinates and y-coordinates from the texture image and are used to map the vertex position with which they are paired.



Example 6
The mouse moves the camera position.

Example 7
The mouse position controls the specular quality of the sphere's material.

Example 8
Many types of lights are simulated. As the box moves with the cursor, it catches light from different sources.

Example 9
Textures are applied to geometry.

Example 6: Camera manipulation (p. 542)
The position and orientation of the camera is set with the `camera()` function. There are nine parameters, arranged in groups of three, to control the camera's position, the location it's pointing to, and the orientation. In this example, the camera stays pointed at the center of a cube, while `mouseY` controls its height. The result is a cube that recedes into the distance when the mouse moves down.

Example 7: Material (p. 543)
The `lightSpecular()` function sets the specular color for lights. The specular quality of a light interacts with the specular material qualities set through the `specular()` function. The specular() function sets the specular color of materials, which sets the color of the highlights. In this example, the parameters to `specular()` change in relation to `mouseX`.

Example 8: Lighting (p. 543)
The functions that create each type of light have different parameters because each light is unique. The `pointLight()` function has six parameters. The first three set the color and the last three set the light's position. The `directionalLight()` function also has six parameters, but they are different. The first three set the color and the last three set the direction the light is pointing. The `spotLight()` function is the most complicated, with eleven parameters to set the color, position, direction, angle, and concentration. This example demonstrates each of these lights as seen through their reflection off a cube. Lights are always reset at the end of `draw()` and need to be recalculated each time through the function.

Example 9: Texture mapping (p. 544)
This example shows how to apply a texture to a flat surface and how to apply a texture to a series of flat surfaces to create a curved shape. The `texture()` function sets the texture that is applied through the `vertex()` function. A version of `vertex()` with five parameters uses the first three to define the (x,y,z) coordinate and the last two to define the (x,y) coordinate of the texture image that maps to this point in 3D. The sine and cosine values that define the geometry to which the texture is applied are predefined within `setup()` so they don't have to be recalculated each time through `draw()`.

Tools for 3D

The computational engines that perform most of the work of transforming 3D scenes into 2D representations on modern computers are either software running on a computer's CPU or specialized graphics processing units (GPUs), the processors on graphics cards. There is a race among the top manufacturers of graphics chipsets—driven largely by the demands of the video-game industry—to produce the fastest and most highly featured hardware renderers that relieve the computer's central processor of most of the work of representing real-time 3D graphics. It is not uncommon for a cheap

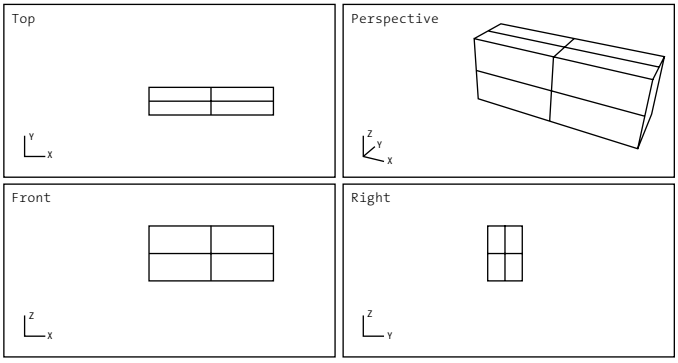
GPU to be more powerful than the main processor on its host machine. For instance, gaming consoles often use advanced GPUs alongside CPUs that are just adequate.

At this stage in the development of consumer 3D graphics, there are only two major standards for the description of 3D scenes to graphics hardware. One, Direct3D, is proprietary to Microsoft and powers its gaming consoles. The other, OpenGL, is an open standard that spun off of work from Silicon Graphics in the early 1990s and is now maintained by a large consortium of companies and industry groups including Apple Computer, IBM, ATI, nVidia, and Sun Microsystems. Many of the 3D language elements implemented in Processing were influenced by OpenGL, and the OpenGL renderer for Processing maps the commands from the Processing API into OpenGL commands.

Befitting their roles in industrial design, engineering, and architecture, today's CAD packages such as AutoCAD, Rhino, and Solidworks focus on precision and constraints in the formation of their geometry. The descriptions they produce are suitable as technical documents or even as the basis for 3D production. An important development in CAD has been the advent of *parametric* design, in which designers are allowed to express abstract relationships between elements that remain invariant even as they change other parts of the design. For example, an architect may specify that an opening in a wall is to be a quarter of the length of the wall. If she later changes the length of the wall, the opening changes size as well. This becomes truly powerful when multiple relationships are established. The length of the wall, for example, may somehow be tied to the path of the sun, and the size of the opening will simply follow. Parametric design, as offered by high-end CAD packages, is standard in many engineering disciplines, and is now beginning to take hold in architecture with software like Bentley Systems' GenerativeComponents and AutoDesk's Revit.

The level of precision found in CAD applications comes at the cost of sketch-like freedom. 3D graphics for entertainment seldom requires this level of control. Animators tend to choose packages that allow for freer form, such as Maya, 3D Studio Max, or Blender. The designs they produce are not as amenable to analysis as CAD drawings, but they are easier to manipulate for image making.

There is a surprising level of similarity in the interfaces of most major 3D packages. A user is typically offered a multipane view of a scene, in which she can see a top, front, side workplanes, and a perspective projection simultaneously:



When operating in any of the top, front, or side planes, the software maps the mouse position into 3D space on the appropriate plane. When operating in the perspective projection it is harder to pick a reference plane; software varies in its solution to this problem, but a plane will be implicitly or explicitly specified.

There are a few notable exceptions to the standard techniques for 3D interfaces. Takeo Igarashi's research project Teddy (1999) allows a user to sketch freely in 2D. The resulting form is interpreted based on a few simple rules as though it were a sketch of the 2D projection of a bulbous form. Subsequent operations on this form are similarly sketch-based and interpreted. Users can slice, erase, or join portions of 3D objects. Teddy is effective because it operates inside a highly constrained formal vocabulary. It would not work as a CAD tool, but it is a highly convincing and evocative interface for 3D sketching. The commercial software SketchUp uses a somewhat less radical but quite ingenious way to solve the 2D input problem. As soon as a user places a base form into the scene, all other operations are interpreted relative to workplanes implied by the faces of existing objects. For instance, a user can select the surface of a wall, and then subsequent mouse input will be interpreted as a projection onto that wall. This is a particularly convenient assumption for architects since so much of the form that makes sense can be described as extruded forms attached at right angles to others.

Conclusion

3D graphics is far too large a topic to cover thoroughly in such a small space. The goal of this section has been to point out landmarks in the disciplines and bodies of technique that surround 3D graphics so that the interested reader can pursue further research. Processing provides a very good practical foundation for this kind of exploration in interactive 3D environments. A phenomenal amount of commercial and academic activity is occurring in computational 3D, not merely for games but also for medicine, architecture, art, engineering, and industrial design. Almost any field that deals with the physical world has call for computational models of it, and our ability to produce evocative simulated objects and environments is the domain of 3D graphics. Where we will take ourselves in our new artificial worlds—or whether we even retain the power to control them—is the subject of much speculation. There has never been a better time to get involved.

Code

Example 1: Drawing in 3D (Transformation)

```
// Rotate a rectangle around the y-axis and x-axis

void setup() {
  size(400, 400, P3D);
  fill(204);
}

void draw() {
  background(0);
  translate(width/2, height/2, -width);
  rotateY(map(mouseX, 0, width, -PI, PI));
  rotateX(map(mouseY, 0, height, -PI, PI));
  noStroke();
  rect(-200, -200, 400, 400);
  stroke(255);
  line(0, 0, -200, 0, 0, 200);
}
```

Example 2: Drawing in 3D (Lights and 3D Shapes)

```
// Draw a sphere on top of a box and move the coordinates with the mouse
// Press a mouse button to turn on the lights

void setup() {
  size(400, 400, P3D);
}

void draw() {
  background(0);
  if (mousePressed == true) {  // If the mouse is pressed,
    lights();                  // turn on lights
  }
  noStroke();
  pushMatrix();
  translate(mouseX, mouseY, -500);
  rotateY(PI/6);               // Rotate around y-axis
  box(400, 100, 400);          // Draw box
  pushMatrix();
  popMatrix();
  translate(0, -200, 0);       // Position the sphere
  sphere(150);                 // Draw sphere on top of box
  popMatrix();
}
```

Example 3: Constructing 3D form

```
// Draw a cylinder centered on the y-axis, going down from y=0 to y=height.
// The radius at the top can be different from the radius at the bottom,
// and the number of sides drawn is variable.

void setup() {
  size(400, 400, P3D);
}

void draw() {
  background(0);
  lights();
  translate(width/2, height/2);
  rotateY(map(mouseX, 0, width, 0, PI));
  rotateZ(map(mouseY, 0, height, 0, -PI));
  noStroke();
  fill(255, 255, 255);
  translate(0, -40, 0);
  drawCylinder(10, 180, 200, 16);   // Draw a mix between a cylinder and a cone
  //drawCylinder(70, 70, 120, 64);  // Draw a cylinder
  //drawCylinder(0, 180, 200, 4);   // Draw a pyramid
}

void drawCylinder(float topRadius, float bottomRadius, float tall, int sides) {
  float angle = 0;
  float angleIncrement = TWO_PI / sides;
  beginShape(QUAD_STRIP);
  for (int i = 0; i < sides + 1; ++i) {
    vertex(topRadius*cos(angle), 0, topRadius*sin(angle));
    vertex(bottomRadius*cos(angle), tall, bottomRadius*sin(angle));
    angle += angleIncrement;
  }
  endShape();

  // If it is not a cone, draw the circular top cap
  if (topRadius != 0) {
    angle = 0;
    beginShape(TRIANGLE_FAN);
    // Center point
    vertex(0, 0, 0);
    for (int i = 0; i < sides + 1; i++) {
      vertex(topRadius * cos(angle), 0, topRadius * sin(angle));
      angle += angleIncrement;
    }
    endShape();
  }

  // If it is not a cone, draw the circular bottom cap
  if (bottomRadius != 0) {
    angle = 0;
    beginShape(TRIANGLE_FAN);
    // Center point
```

```
    vertex(0, tall, 0);
    for (int i = 0; i < sides+1; i++) {
      vertex(bottomRadius * cos(angle), tall, bottomRadius * sin(angle));
      angle += angleIncrement;
    }
    endShape();
  }
}
```

Example 4: DXF export

```
// Export a DXF file when the R key is pressed

import processing.dxf.*;

boolean record = false;

void setup() {
  size(400, 400, P3D);
  noStroke();
  sphereDetail(12);
}

void draw() {
  if (record == true) {
    beginRaw(DXF, "output.dxf");   // Start recording to the file
  }
  lights();
  background(0);
  translate(width/3, height/3, -200);
  rotateZ(map(mouseY, 0, height, 0, PI));
  rotateY(map(mouseX, 0, width, 0, HALF_PI));
  for (int y = -2; y < 2; y++) {
    for (int x = -2; x < 2; x++) {
      for (int z = -2; z < 2; z++) {
        pushMatrix();
        translate(120*x, 120*y, -120*z);
        sphere(30);
        popMatrix();
      }
    }
  }
  if (record == true) {
    endRaw();
    record = false;                // Stop recording to the file
  }
}

void keyPressed() {
  if (key == 'R' || key == 'r') {  // Press R to save the file
    record = true;
  }
}
```

Example 5: OBJ import

```
// Import and display an OBJ model

import saito.objloader.*;

OBJModel model;

void setup() {
  size(400, 400, P3D);
  model = new OBJModel(this);
  model.load("chair.obj");  // Model must be in the data directory
  model.drawMode(POLYGON);
  noStroke();
}

void draw() {
  background(0);
  lights();
  pushMatrix();
  translate(width/2, height, -width);
  rotateY(map(mouseX, 0, width, -PI, PI));
  rotateX(PI/4);
  scale(6.0);
  model.draw();
  popMatrix();
}
```

Example 6: Camera manipulation

```
// The camera lifts up while looking at the same point

void setup() {
  size(400, 400, P3D);
  fill(204);
}

void draw() {
  lights();
  background(0);
  // Change height of the camera with mouseY
  camera(300.0, mouseY, 220.0,  // eyeX, eyeY, eyeZ
         0.0, 0.0, 0.0,         // centerX, centerY, centerZ
         0.0, 1.0, 0.0);        // upX, upY, upZ
  noStroke();
  box(90);
  stroke(255);
  line(-100, 0, 0, 100, 0, 0);
  line(0, -100, 0, 0, 100, 0);
  line(0, 0, -100, 0, 0, 100);
}
```

Example 7: Material

```
// Vary the specular reflection component of a material
// with vertical position of the mouse

void setup() {
  size(400, 400, P3D);
  noStroke();
  colorMode(RGB, 1);
  fill(0.4);
}

void draw() {
  background(0);
  translate(width/2, height/2);
  // Set the specular color of lights that follow
  lightSpecular(1, 1, 1);
  directionalLight(0.8, 0.8, 0.8, 0, 0, -1);
  float s = mouseX / float(width);
  specular(s, s, s);
  sphere(100);
}
```

Example 8: Lighting

```
// Draw a box with three different kinds of lights

void setup() {
  size(400, 400, P3D);
  noStroke();
}

void draw()  {
  background(0);
  translate(width/2, height/2);
  // Orange point light on the right
  pointLight(150, 100, 0,                   // Color
             200, -150, 0);                 // Position
  // Blue directional light from the left
  directionalLight(0, 102, 255,             // Color
                   1, 0, 0);                // The x-, y-, z-axis direction
  // Yellow spotlight from the front
  spotLight(255, 255, 109,                  // Color
            0, 40, 200,                     // Position
            0, -0.5, -0.5,                  // Direction
            PI/2, 2);                       // Angle, concentration

  rotateY(map(mouseX, 0, width, 0, PI));
  rotateX(map(mouseY, 0, height, 0, PI));
  box(200);
}
```

Example 9: Texture mapping

```
// Load an image and draw it onto a cylinder and a quad

int tubeRes = 32;
float[] tubeX = new float[tubeRes];
float[] tubeY = new float[tubeRes];

PImage img;

void setup() {
  size(400, 400, P3D);
  img = loadImage("berlin-1.jpg");
  float angle = 270.0 / tubeRes;
  for (int i = 0; i < tubeRes; i++) {
    tubeX[i] = cos(radians(i * angle));
    tubeY[i] = sin(radians(i * angle));
  }
  noStroke();
}

void draw() {
  background(0);
  translate(width/2, height/2);
  rotateX(map(mouseY, 0, height, -PI, PI));
  rotateY(map(mouseX, 0, width, -PI, PI));
  beginShape(QUAD_STRIP);
  texture(img);
  for (int i = 0; i < tubeRes; i++) {
    float x = tubeX[i] * 100;
    float z = tubeY[i] * 100;
    float u = img.width / tubeRes * i;
    vertex(x, -100, z, u, 0);
    vertex(x, 100, z, u, img.height);
  }
  endShape();
  beginShape(QUADS);
  texture(img);
  vertex(0, -100, 0, 0, 0);
  vertex(100,-100, 0, 100, 0);
  vertex(100, 100, 0, 100, 100);
  vertex(0, 100, 0, 0, 100);
  endShape();
}
```

Resources

Books and online resources

Hearn, Donald, and M. Pauline Baker. *Computer Graphics: C Version*. Second edition. Upper Saddle Prentice Hall, 1986.

Foley, James D., and Andries van Dam et al. *Computer Graphics: Principles and Practice in C*. Second edition. Addison-Wesley, 1995.

Greenwold, Simon. "Spatial Computing." Master's thesis, MIT Media Lab, 2003. http://acg.media.mit.edu/people/simong.

OpenGL Architecture Review Board. *OpenGL Programming Guide*. Fourth edition. Addison-Wesley, 2003. An earlier edition is available free, online at http://www.opengl.org/documentation/red_book.

OpenGL Architecture Review Board. *OpenGL Reference Manual*. Fourth edition. Addison-Wesley, 2004. An earlier edition is available free, online at http://www.opengl.org/documentation/blue_book.

Silicon Graphics Inc. (SGI). Computer graphics pioneer. http://en.wikipedia.org/wiki/Silicon_Graphics.

Wotsit's Format. Web resource documenting file formats. http://www.wotsit.org.

Software

Blender. Open source 3D modeling and animation software. http://www.blender.org.

OpenGL. 3D Graphics format. http://www.opengl.org, http://en.wikipedia.org/wiki/Opengl.

DirectX. Microsoft's 3D Graphics format. http://www.microsoft.com/windows/directx.

Z Corporation. Manufacturer of 3D printers. http://www.zcorp.com.

AutoCAD. 2D, 3D drafting software. http://www.autodesk.com/autocad, http://en.wikipedia.org/wiki/AutoCAD.

Rivit Building. Building information modeling software. http://www.autodesk.com/revitbuilding.

DXF. Widely supported 3D file format introduced in AutoCAD 1.0. http://www.autodesk.com/dxf/, http://en.wikipedia.org/wiki/DXF.

OBJ. Open 3D file format. http://en.wikipedia.org/wiki/Obj.

Rhino. 3D modeling software. http://www.rhino3d.com.

Solidworks. 3D modeling software. http://www.solidworks.com.

Maya. 3D modeling and animation software. http://www.autodesk.com/maya.

3D Studio Max. 3D modeling and animation software. http://www.autodesk.com/3dsmax.

GenerativeComponents. 3D Parametric design software. http://www.bentley.com.

Teddy: 3D Freeform Sketching. http://www-ui.is.s.u-tokyo.ac.jp/~takeo/teddy/teddy.htm.

SketchUp. http://www.sketchup.com.

Artworks and games

Microsoft. Flight Simulator. Documented at Flight Simulator History: http://fshistory.simflight.com/fsh/index.htm.

Sierra Entertainment. King's Quest. Documented at Vintage-Sierra: http://www.vintage-sierra.com/kingsquest.html.

id Software. Wolfenstein 3D. http://en.wikipedia.org/wiki/Wolfenstein_3D.

Davies, Char. Osmose. 1995. http://www.immersence.com.

Greenwold, Simon. Installation. 2004. http://acg.media.mit.edu/people/simong/installationNew/cover.html.

Rees, Michael. Large Small and Moving. 2004. http://www.michaelrees.com/sacksmo/catalogue.html.

#2
#5
#4
#3

Extension 3: Vision

Text by Golan Levin

A well-known anecdote relates how, sometime in 1966, the legendary artificial intelligence pioneer Marvin Minsky directed an undergraduate student to solve "the problem of computer vision" as a summer project.[1] This anecdote is often resuscitated to illustrate how egregiously the difficulty of computational vision has been underestimated. Indeed, nearly forty years later the discipline continues to confront numerous unsolved (and perhaps unsolvable) challenges, particularly with respect to high-level "image understanding" issues such as pattern recognition and feature recognition. Nevertheless, the intervening decades of research have yielded a great wealth of well-understood, low-level techniques that are able, under controlled circumstances, to extract meaningful information from a camera scene. These techniques are indeed elementary enough to be implemented by novice programmers at the undergraduate or even high-school level.

Computer vision in interactive art

The first interactive artwork to incorporate computer vision was, interestingly enough, also one of the first interactive artworks. Myron Krueger's legendary *Videoplace*, developed between 1969 and 1975, was motivated by his deeply felt belief that the entire human body ought to have a role in our interactions with computers. In the Videoplace installation, a participant stands in front of a backlit wall and faces a video projection screen. The participant's silhouette is then digitized and its posture, shape, and gestural movements analyzed. In response, *Videoplace* synthesizes graphics such as small "critters" that climb up the participant's projected silhouette, or colored loops drawn between the participant's fingers. Krueger also allowed participants to paint lines with their fingers, and, indeed, entire shapes with their bodies; eventually, *Videoplace* offered more than fifty compositions and interactions. *Videoplace* is notable for many "firsts" in the history of human-computer interaction. Some of its interaction modules allowed two participants in mutually remote locations to participate in the same shared video space, connected across the network—an implementation of the first multiperson virtual reality, or, as Krueger termed it, an "artificial reality." *Videoplace*, it should be noted, was developed before the mouse became the ubiquitous desktop device it is today, and was (in part) created to demonstrate interface alternatives to the keyboard terminals that dominated computing so completely in the early 1970s.

Messa di Voce (p. 511), created by this text's author in collaboration with Zachary Lieberman, uses whole-body vision-based interactions similar to Krueger's, but combines them with speech analysis and situates them within a kind of projection-based

Rafael Lozano-Hemmer. *Standards and Double Standards*, 2004. Images courtesy of the bitforms gallery, nyc.

augmented reality. In this audiovisual performance, the speech, shouts, and songs produced by two abstract vocalists are visualized and augmented in real time by synthetic graphics. To accomplish this, a computer uses a set of vision algorithms to track the locations of the performers' heads; this computer also analyzes the audio signals coming from the performers' microphones. In response, the system displays various kinds of visualizations on a projection screen located just behind the performers; these visualizations are synthesized in ways that are tightly coupled to the sounds being spoken and sung. With the help of the head-tracking system, moreover, these visualizations are projected such that they appear to emerge directly from the performers' mouths.

Rafael Lozano-Hemmer's installation *Standards and Double Standards* (2004) incorporates full-body input in a less direct, more metaphorical context. This work consists of fifty leather belts, suspended at waist height from robotic servomotors mounted on the ceiling of the exhibition room. Controlled by a computer vision-based tracking system, the belts rotate automatically to follow the public, turning their buckles slowly to face passers-by. Lozano-Hemmer's piece "turns a condition of pure surveillance into an 'absent crowd' using a fetish of paternal authority: the belt."[2]

The theme of surveillance plays a foreground role in David Rokeby's *Sorting Daemon* (2003). Motivated by the artist's concerns about the increasing use of automated systems for profiling people as part of the "war on terrorism," this site-specific installation works toward the automatic construction of a diagnostic portrait of its social (and racial) environment. Rokeby writes: "The system looks out onto the street, panning, tilting and zooming, looking for moving things that might be people. When it finds what it thinks might be a person, it removes the person's image from the background. The extracted person is then divided up according to areas of similar colour. The resulting swatches of colour are then organized [by hue, saturation and size] within the arbitrary context of the composite image" projected onsite at the installation's host location.[3]

Another project themed around issues of surveillance is *Suicide Box*, by the Bureau of Inverse Technology (Natalie Jeremijenko and Kate Rich). Presented as a device for measuring the hypothetical "despondency index" of a given locale, the *Suicide Box* nevertheless records very real data regarding suicide jumpers from the Golden Gate Bridge. According to the artists, "The *Suicide Box* is a motion-detection video system, positioned in range of the Golden Gate Bridge, San Francisco, in 1996. It watched the bridge constantly and when it recognized vertical motion, captured it to a video record. The resulting footage displays as a continuous stream the trickle of people who jump off the bridge. The Golden Gate Bridge is the premiere suicide destination in the United States; a 100-day initial deployment period of the *Suicide Box* recorded 17 suicides. During the same time period the Port Authority counted only 13."[4] Elsewhere, Jeremijenko has explained that "the idea was to track a tragic social phenomenon which was not being counted—that is, doesn't count."[5] The *Suicide Box* has met with considerable controversy, ranging from ethical questions about recording the suicides to disbelief that the recordings could be real. Jeremijenko, whose aim is to address the hidden politics of technology, has pointed out that such attitudes express a recurrent theme—"the inherent suspicion of artists working with material evidence"—evidence

obtained, in this case, with the help of machine vision-based surveillance.

Considerably less macabre is Christian Möller's clever Cheese installation (2003), which the artist developed in collaboration with the California Institute of Technology and the Machine Perception Laboratories of the University of California, San Diego. Motivated, perhaps, by the culture shock of his relocation to Hollywood, the German-born Möller directed "six actresses to hold a smile for as long as they could, up to one and a half hours. Each ongoing smile is scrutinized by an emotion recognition system, and whenever the display of happiness fell below a certain threshold, an alarm alerted them to show more sincerity."[6] The installation replays recordings of the analyzed video on six flat-panel monitors, with the addition of a fluctuating graphic level-meter to indicate the strength of each actress' smile. The technical implementation of this artwork's vision-based emotion recognition system is quite sophisticated.

As can be seen from these examples, artworks employing computer vision range from the highly formal and abstract to the humorous and sociopolitical. They concern themselves with the activities of willing participants, paid volunteers, or unaware strangers. They track people of interest at a wide variety of spatial scales, from extremely intimate studies of their facial expressions, to the gestures of their limbs, to the movements of entire bodies. The examples above represent just a small selection of notable works in the field and of the ways in which people (and objects) have been tracked and dissected by video analysis. Other noteworthy artworks that use machine vision include Marie Sester's *Access*; Joachim Sauter and Dirk Lüsebrink's *Zerseher* and *Bodymover*; Scott Snibbe's *Boundary Functions* and *Screen Series*; Camille Utterback and Romy Achituv's *TextRain*; Jim Campbell's *Solstice*; Christa Sommerer and Laurent Mignonneau's *A-Volve*; Danny Rozin's *Wooden Mirror*; Chico MacMurtrie's *Skeletal Reflection*, and various works by Simon Penny, Toshio Iwai, and numerous others. No doubt many more vision-based artworks remain to be created, especially as these techniques gradually become incorporated into developing fields like physical computing and robotics.

Elementary computer vision techniques

To understand how novel forms of interactive media can take advantage of computer vision techniques, it is helpful to begin with an understanding of the kinds of problems that vision algorithms have been developed to address, and of their basic mechanisms of operation. The fundamental challenge presented by digital video is that it is computationally "opaque." Unlike text, digital video data in its basic form—stored solely as a stream of rectangular pixel buffers—contains no intrinsic semantic or symbolic information. There is no widely agreed upon standard for representing the content of video, in a manner analogous to HTML, XML, or even ASCII for text (though some new initiatives, notably the MPEG-7 description language, may evolve into such a standard in the future). As a result, a computer, without additional programming, is unable to answer even the most elementary questions about whether a video stream contains a person or object, or whether an outdoor video scene shows daytime or nighttime, et

cetera. The discipline of computer vision has developed to address this need.

Many low-level computer vision algorithms are geared to the task of distinguishing which pixels, if any, belong to people or other objects of interest in the scene. Three elementary techniques for accomplishing this are frame differencing, which attempts to locate features by detecting their movements; background subtraction, which locates visitor pixels according to their difference from a known background scene; and brightness thresholding, which uses hoped-for differences in luminosity between foreground people and their background environment. These algorithms, described in the following examples, are extremely simple to implement and help constitute a base of detection schemes from which sophisticated interactive systems may be built.

Example 1: Detecting motion (p. 556)
The movements of people (or other objects) within the video frame can be detected and quantified using a straightforward method called frame differencing. In this technique, each pixel in a video frame F1 is compared with its corresponding pixel in the subsequent frame F2. The difference in color and/or brightness between these two pixels is a measure of the amount of movement in that particular location. These differences can be summed across all of the pixels' locations to provide a single measurement of the aggregate movement within the video frame. In some motion detection implementations, the video frame is spatially subdivided into a grid of cells, and the values derived from frame differencing are reported for each of the individual cells. For accuracy, the frame differencing algorithm depends on relatively stable environmental lighting, and on having a stationary camera (unless it is the motion of the camera that is being measured).

Example 2: Detecting presence (p. 557)
A technique called background subtraction makes it possible to detect the presence of people or other objects in a scene, and to distinguish the pixels that belong to them from those that do not. The technique operates by comparing each frame of video with a stored image of the scene's background, captured at a point in time when the scene was known to be empty. For every pixel in the frame, the absolute difference is computed between its color and that of its corresponding pixel in the stored background image; areas that are very different from the background are likely to represent objects of interest. Background subtraction works well in heterogeneous environments, but it is very sensitive to changes in lighting conditions and depends on objects of interest having sufficient contrast against the background scene.

Example 3: Detection through brightness thresholding (p. 559)
With the aid of controlled illumination (such as backlighting) and/or surface treatments (such as high-contrast paints), it is possible to ensure that objects are considerably darker or lighter than their surroundings. In such cases objects of interest can be distinguished based on their brightness alone. To do this, each video pixel's brightness is compared to a threshold value and tagged accordingly as foreground or background.

Example 1. *Detects motion by comparing each video frame to the previous frame. The change is visualized and is calculated as a number.*

Example 2. *Detects the presence of someone or something in front of the camera by comparing each video frame with a previously saved frame. The change is visualized and is calculated as a number.*

Example 3. *Distinguishes the silhouette of people or objects in each video frame by comparing each pixel to a threshold value. The circle is filled with white when it is within the silhouette.*

Example 4. *Tracks the brightest object in each video frame by calculating the brightest pixel. The light from the flashlight is the brightest element in the frame; therefore, the circle follows it.*

Example 4: Brightness tracking (p. 560)
A rudimentary scheme for object tracking, ideal for tracking the location of a single illuminated point (such as a flashlight), finds the location of the single brightest pixel in every fresh frame of video. In this algorithm, the brightness of each pixel in the incoming video frame is compared with the brightest value yet encountered in that frame; if a pixel is brighter than the brightest value yet encountered, then the location and brightness of that pixel are stored. After all of the pixels have been examined, then the brightest location in the video frame is known. This technique relies on an operational assumption that there is only one such object of interest. With trivial modifications, it can equivalently locate and track the darkest pixel in the scene, or track multiple and differently colored objects.

Of course, many more software techniques exist, at every level of sophistication, for detecting, recognizing, and interacting with people and other objects of interest. Each of the tracking algorithms described above, for example, can be found in elaborated versions that amend its various limitations. Other easy-to-implement algorithms can compute specific features of a tracked object, such as its area, center of mass, angular orientation, compactness, edge pixels, and contour features such as corners and cavities. On the other hand, some of the most difficult to implement algorithms, representing the cutting edge of computer vision research today, are able (within limits) to recognize unique people, track the orientation of a person's gaze, or correctly identify facial expressions. Pseudocodes, source codes, or ready-to-use implementations of all of these techniques can be found on the Internet in excellent resources like Daniel Huber's Computer Vision Homepage, Robert Fisher's HIPR (Hypermedia Image Processing Reference), or in the software toolkits discussed on pages 554-555.

Computer vision in the physical world

Unlike the human eye and brain, no computer vision algorithm is completely general, which is to say, able to perform its intended function given any possible video input. Instead, each software tracking or detection algorithm is critically dependent on certain unique assumptions about the real-world video scene it is expected to analyze. If any of these expectations are not met, then the algorithm can produce poor or ambiguous results or even fail altogether. For this reason, it is essential to design physical conditions in tandem with the development of computer vision code, and to select the software techniques that are most compatible with the available physical conditions.

Background subtraction and brightness thresholding, for example, can fail if the people in the scene are too close in color or brightness to their surroundings. For these algorithms to work well, it is greatly beneficial to prepare physical circumstances that naturally emphasize the contrast between people and their environments. This can be achieved with lighting situations that silhouette the people, or through the use of specially colored costumes. The frame-differencing technique, likewise, fails to detect people if they are stationary. It will therefore have very different degrees of success

detecting people in videos of office waiting rooms compared with videos of the Tour de France bicycle race.

A wealth of other methods exist for optimizing physical conditions in order to enhance the robustness, accuracy, and effectiveness of computer vision software. Most are geared toward ensuring a high-contrast, low-noise input image. Under low-light conditions, for example, one of the most helpful such techniques is the use of infrared (IR) illumination. Infrared, which is invisible to the human eye, can supplement the light detected by conventional black-and-white security cameras. Using IR significantly improves the signal-to-noise ratio of video captured in low-light circumstances and can even permit vision systems to operate in (apparently) complete darkness. Another physical optimization technique is the use of retroreflective marking materials, such as those manufactured by 3M Corporation for safety uniforms. These materials are remarkably efficient at reflecting light back toward their source of illumination and are ideal aids for ensuring high-contrast video of tracked objects. If a small light is placed coincident with the camera's axis, objects with retroreflective markers will be detected with tremendous reliability.

Finally, some of the most powerful physical optimizations for machine vision can be made without intervening in the observed environment at all, through well-informed selections of the imaging system's camera, lens, and frame-grabber components. To take one example, the use of a "telecentric" lens can significantly improve the performance of certain kinds of shape-based or size-based object recognition algorithms. For this type of lens, which has an effectively infinite focal length, magnification is nearly independent of object distance. As one manufacturer describes it, "an object moved from far away to near the lens goes into and out of sharp focus, but its image size is constant. This property is very important for gauging three-dimensional objects, or objects whose distance from the lens is not known precisely."[7] Likewise, polarizing filters offer a simple, nonintrusive solution to another common problem in video systems, namely glare from reflective surfaces. And a wide range of video cameras are available, optimized for conditions like high-resolution capture, high-frame-rate capture, short exposure times, dim light, ultraviolet light, and thermal imaging. It pays to research imaging components carefully.

As we have seen, computer vision algorithms can be selected to negotiate best the physical conditions presented by the world, and physical conditions can be modified to be more easily legible to vision algorithms. But even the most sophisticated algorithms and the highest-quality hardware cannot help us find meaning where there is none, or track an object that cannot be described in code. It is therefore worth emphasizing that some visual features contain more information about the world, and are also more easily detected by the computer, than others. In designing systems to "see for us," we must not only become freshly awakened to the many things about the world that make it visually intelligible to us, but also develop a keen intuition about their ease of computability. The sun is the brightest point in the sky, and by its height also indicates the time of day. The mouth cavity is easily segmentable as a dark region, and the circularity of its shape is also closely linked to vowel sound. The pupils of the eyes emit an easy-to-track infrared retroreflection, and they also indicate a person's direction of

gaze. Simple frame differencing makes it easy to track motion in a video. The *Suicide Box* (p. 548) uses this technique to dramatic effect.

Tools for computer vision

It can be a rewarding experience to implement machine vision techniques from scratch using code such as the examples provided in this section. To make this possible, the only requirement of one's software development environment is that it should provide direct read-access to the array of video pixels obtained by the computer's frame-grabber. Hopefully, the example algorithms discussed earlier illustrate that creating low-level vision algorithms from first principles isn't so hard. Of course, a vast range of functionality can also be obtained immediately from readily available solutions. Some of the most popular machine vision toolkits take the form of plug-ins or extension libraries for commercial authoring environments geared toward the creation of interactive media. Such plug-ins simplify the developer's problem of connecting the results of the vision-based analysis to the audio, visual, and textual affordances generally provided by such authoring systems.

Many vision plug-ins have been developed for Max/MSP/Jitter, a visual programming environment that is widely used by electronic musicians and VJs. Originally developed at the Parisian IRCAM research center in the mid-1980s and now marketed commercially by the California-based Cycling'74 company, this extensible environment offers powerful control of (and connectivity between) MIDI devices, real-time sound synthesis and analysis, OpenGL-based 3D graphics, video filtering, network communications, and serial control of hardware devices. The various computer vision plug-ins for Max/MSP/Jitter, such as David Rokeby's SoftVNS, Eric Singer's Cyclops, and Jean-Marc Pelletier's CV.Jit, can be used to trigger any Max processes or control any system parameters. Pelletier's toolkit, which is the most feature-rich of the three, is also the only one that is freeware. CV.Jit provides abstractions to assist users in tasks such as image segmentation, shape and gesture recognition, and motion tracking, as well as educational tools that outline the basics of computer vision techniques.

Some computer vision toolkits take the form of stand-alone applications and are designed to communicate the results of their analyses to other environments (such as Processing, Director, or Max) through protocols like MIDI, serial RS-232, UDP, or TCP/IP networks. BigEye, developed by the STEIM (Studio for Electro-Instrumental Music) group in Holland, is a simple and inexpensive example. BigEye can track up to 16 objects of interest simultaneously, according to their brightness, color, and size. The software allows for a simple mode of operation in which the user can quickly link MIDI messages to many object parameters, such as position, speed, and size. Another example is the powerful EyesWeb open platform, a free system developed at the University of Genoa. Designed with a special focus on the analysis and processing of expressive gesture, EyesWeb includes a collection of modules for real-time motion tracking and extraction of movement cues from human full-body movement; a collection of modules for analysis of occupation of 2D space; and a collection of modules for extraction of features from

trajectories in 2D space. EyesWeb's extensive vision affordances make it highly recommended for students.

The most sophisticated toolkits for computer vision generally demand greater familiarity with digital signal processing, and they require developers to program in compiled languages like C++ rather than languages like Java, Lingo, or Max. The Intel Integrated Performance Primitives (IPP) library, for example, is among the most general commercial solutions available for computers with Intel-based CPUs. The OpenCV library, by contrast, is a free, open source toolkit with nearly similar capabilities and a tighter focus on commonplace computer vision tasks. The capabilities of these tools, as well as all of those mentioned above, are continually evolving.

Processing includes a basic video library that handles getting pixel information from a camera or movie file as demonstrated in the examples included with this text. The computer vision capabilities of Processing are extended by libraries like Myron, which handles video input and has basic image processing capabilities. Other libraries connect Processing to EyesWeb and OpenCV. They can be found on the libraries page of the Processing website: *www.processing.org/reference/libraries*.

Conclusion

Computer vision algorithms are increasingly used in interactive and other computer-based artworks to track people's activities. Techniques exist that can create real-time reports about people's identities, locations, gestural movements, facial expressions, gait characteristics, gaze directions, and other attributes. Although the implementation of some vision algorithms requires advanced understanding of image processing and statistics, a number of widely used and highly effective techniques can be implemented by novice programmers in as little as an afternoon. For artists and designers who are familiar with popular multimedia authoring systems like Macromedia Director and Max/MSP/Jitter, a wide range of free and commercial toolkits are also available that provide ready access to more advanced vision functionalities.

Since the reliability of computer vision algorithms is limited according to the quality of the incoming video scene and the definition of a scene's quality is determined by the specific algorithms that are used to analyze it, students approaching computer vision for the first time are encouraged to apply as much effort to optimizing their physical scenario as they do their software code. In many cases, a cleverly designed physical environment can permit the tracking of phenomena that might otherwise require much more sophisticated software. As computers and video hardware become more available, and software-authoring tools continue to improve, we can expect to see the use of computer vision techniques increasingly incorporated into media-art education and into the creation of games, artworks, and many other applications.

Notes

1. http://mechanism.ucsd.edu/~bill/research/mercier/2ndlecture.pdf.
2. http://www.fundacion.telefonica.com/at/rlh/eproyecto.html.

3. http://homepage.mac.com/davidrokeby/sorting.html.
4. http://www.bureauit.org/sbox.
5. http://www.wired.com/news/culture/0,1284,64720,00.html.
6. http://www.christian-moeller.com.
7. http://www.mellesgriot.com/pdf/pg11-19.pdf.

Code

Video can be captured into Processing from USB cameras, IEEE 1394 cameras, or video cards with composite or S-video input devices. The examples that follow assume you already have a camera working with Processing. Before trying these examples, first get the examples included with the Processing software to work. Sometimes you can plug a camera into your computer and it will work immediately. Other times it's a difficult process involving trial-and-error changes. It depends on the operating system, the camera, and how the computer is configured. For the most up-to-date information, refer to the Video reference on the Processing website: *www.processing.org/reference/libraries*.

Example 1: Detecting motion

```
// Quantify the amount of movement in the video frame using frame-differencing

import processing.video.*;

int numPixels;
int[] previousFrame;
Capture video;

void setup(){
  size(640, 480);  // Change size to 320 x 240 if too slow at 640 x 480
  video = new Capture(this, width, height, 24);
  numPixels = video.width * video.height;
  // Create an array to store the previously captured frame
  previousFrame = new int[numPixels];
}

void draw() {
  if (video.available()) {
    // When using video to manipulate the screen, use video.available() and
    // video.read() inside the draw() method so that it's safe to draw to the screen
    video.read();        // Read the new frame from the camera
    video.loadPixels();  // Make its pixels[] array available

    int movementSum = 0; // Amount of movement in the frame
    loadPixels();

    for (int i = 0; i < numPixels; i++) {  // For each pixel in the video frame...
```

```
      color currColor = video.pixels[i];
      color prevColor = previousFrame[i];

      // Extract the red, green, and blue components from current pixel
      int currR = (currColor >> 16) & 0xFF;  // Like red(), but faster (see p. 673)
      int currG = (currColor >> 8) & 0xFF;
      int currB =  currColor & 0xFF;

      // Extract red, green, and blue components from previous pixel
      int prevR = (prevColor >> 16) & 0xFF;
      int prevG = (prevColor >> 8) & 0xFF;
      int prevB = prevColor & 0xFF;

      // Compute the difference of the red, green, and blue values
      int diffR = abs(currR - prevR);
      int diffG = abs(currG - prevG);
      int diffB = abs(currB - prevB);

      // Add these differences to the running tally
      movementSum += diffR + diffG + diffB;
      // Render the difference image to the screen
      pixels[i] = color(diffR, diffG, diffB);
      // The following line is much faster, but more confusing to read
      //pixels[i] = 0xff000000 | (diffR << 16) | (diffG << 8) | diffB;
      // Save the current color into the 'previous' buffer
      previousFrame[i] = currColor;
    }

    // To prevent flicker from frames that are all black (no movement),
    // only update the screen if the image has changed.
    if (movementSum > 0) {
      updatePixels();
      println(movementSum);  // Print the total amount of movement to the console
    }
  }
}
```

Example 2: Detecting presence

```
// Detect the presence of people and objects in the frame using a simple
// background-subtraction technique. To initialize the background, press a key.

import processing.video.*;

int numPixels;
int[] backgroundPixels;
Capture video;

void setup() {
  size(640, 480);  // Change size to 320 x 240 if too slow at 640 x 480
  video = new Capture(this, width, height, 24);
  numPixels = video.width * video.height;
```

```
  // Create array to store the background image
  backgroundPixels = new int[numPixels];
  // Make the pixels[] array available for direct manipulation
  loadPixels();
}

void draw() {
  if (video.available()) {
    video.read();        // Read a new video frame
    video.loadPixels();  // Make the pixels of video available

    // Difference between the current frame and the stored background
    int presenceSum = 0;

    for (int i = 0; i < numPixels; i++) {  // For each pixel in the video frame...
      // Fetch the current color in that location, and also the color
      // of the background in that spot
      color currColor = video.pixels[i];
      color bkgdColor = backgroundPixels[i];

      // Extract the red, green, and blue components of the current pixel's color
      int currR = (currColor >> 16) & 0xFF;
      int currG = (currColor >> 8) & 0xFF;
      int currB = currColor & 0xFF;

      // Extract the red, green, and blue components of the background pixel's color
      int bkgdR = (bkgdColor >> 16) & 0xFF;
      int bkgdG = (bkgdColor >> 8) & 0xFF;
      int bkgdB = bkgdColor & 0xFF;

      // Compute the difference of the red, green, and blue values
      int diffR = abs(currR - bkgdR);
      int diffG = abs(currG - bkgdG);
      int diffB = abs(currB - bkgdB);

      // Add these differences to the running tally
      presenceSum += diffR + diffG + diffB;
      // Render the difference image to the screen
      pixels[i] = color(diffR, diffG, diffB);
      // The following line does the same thing much faster, but is more technical
      //pixels[i] = 0xFF000000 | (diffR << 16) | (diffG << 8) | diffB;
    }
    updatePixels();        // Notify that the pixels[] array has changed
    println(presenceSum);  // Print out the total amount of movement
  }
}

// When a key is pressed, capture the background image into the backgroundPixels
// buffer by copying each of the current frame's pixels into it.
void keyPressed() {
  video.loadPixels();
  arraycopy(video.pixels, backgroundPixels);
}
```

Example 3: Detection through brightness thresholding

```
// Determines whether a test location (such as the cursor) is contained within
// the silhouette of a dark object

import processing.video.*;

color black = color(0);
color white = color(255);
int numPixels;
Capture video;

void setup() {
  size(640, 480);  // Change size to 320 x 240 if too slow at 640 x 480
  strokeWeight(5);
  video = new Capture(this, width, height, 24);
  numPixels = video.width * video.height;
  noCursor();
  smooth();
}

void draw() {
  if (video.available()) {
    video.read();
    video.loadPixels();
    int threshold = 127;    // Set the threshold value
    float pixelBrightness;  // Declare variable to store a pixel's color

    // Turn each pixel in the video frame black or white depending on its brightness
    loadPixels();
    for (int i = 0; i < numPixels; i++) {
      pixelBrightness = brightness(video.pixels[i]);
      if (pixelBrightness > threshold) {  // If the pixel is brighter than the
        pixels[i] = white;                // threshold value, make it white
      } else {                            // Otherwise,
        pixels[i] = black;                // make it black
      }
    }
    updatePixels();

    // Test a location to see where it is contained. Fetch the pixel at the test
    // location (the cursor), and compute its brightness
    int testValue = get(mouseX, mouseY);
    float testBrightness = brightness(testValue);
    if (testBrightness > threshold) {     // If the test location is brighter than
      fill(black);                        // the threshold set the fill to black
    } else {                              // Otherwise,
      fill(white);                        // set the fill to white
    }
    ellipse(mouseX, mouseY, 20, 20);
  }
}
```

Example 4: Brightness tracking

```
// Tracks the brightest pixel in a live video signal

import processing.video.*;

Capture video;

void setup(){
  size(640, 480);  // Change size to 320 x 240 if too slow at 640 x 480
  video = new Capture(this, width, height, 30);
  noStroke();
  smooth();
}

void draw() {
  if (video.available()) {
    video.read();
    image(video, 0, 0, width, height);  // Draw the webcam video onto the screen

    int brightestX = 0;        // X-coordinate of the brightest video pixel
    int brightestY = 0;        // Y-coordinate of the brightest video pixel
    float brightestValue = 0;  // Brightness of the brightest video pixel

    // Search for the brightest pixel: For each row of pixels in the video image and
    // for each pixel in the yth row, compute each pixel's index in the video
    video.loadPixels();
    int index = 0;
    for (int y = 0; y < video.height; y++) {
      for (int x = 0; x < video.width; x++) {
        // Get the color stored in the pixel
        int pixelValue = video.pixels[index];
        // Determine the brightness of the pixel
        float pixelBrightness = brightness(pixelValue);
        // If that value is brighter than any previous, then store the
        // brightness of that pixel, as well as its (x,y) location
        if (pixelBrightness > brightestValue){
          brightestValue = pixelBrightness;
          brightestY = y;
          brightestX = x;
        }
        index++;
      }
    }

    // Draw a large, yellow circle at the brightest pixel
    fill(255, 204, 0, 128);
    ellipse(brightestX, brightestY, 200, 200);
  }
}
```

Resources

Computer vision software toolkits

Camurri, Antonio, et al. Eyesweb. Vision-oriented software development environment. http://www.eyesweb.org.

Cycling'74 Inc. Max/MSP/Jitter. Graphic software development environment. http://www.cycling74.com.

Davies, Bob, et al. OpenCV. Open source computer vision library. http://sourceforge.net/projects/opencvlibrary.

Nimoy, Joshua. Myron (WebCamXtra). Library (plug-in) for Macromedia Director and Processing. http://webcamxtra.sourceforge.net.

Pelletier, Jean-Marc. CV.Jit. Extension library for Max/MSP/Jitter. http://www.iamas.ac.jp/~jovano2/cv.

Rokeby, David. SoftVNS. Extension library for Max/MSP/Jitter. http://homepage.mac.com/davidrokeby/softVNS.html.

Singer, Eric. Cyclops. Extension library for Max/MSP/Jitter. http://www.cycling74.com/products/cyclops.html.

STEIM (Studio for Electro-Instrumental Music). BigEye. Video analysis software. http://www.steim.org

Texts and artworks

Bureau of Inverse Technology. *Suicide Box*. http://www.bureauit.org/sbox.

Bechtel, William. The Cardinal Mercier Lectures at the Catholic University of Louvain. Lecture 2, An Exemplar. Neural Mechanism: The Brain's Visual Processing System. 2003, p.1. http://mechanism.ucsd.edu/~bill/research/mercier/2ndlecture.pdf.

Fisher, Robert, et. al. HIPR (The Hypermedia Image Processing Reference). http://homepages.inf.ed.ac.uk/rbf/HIPR2/index.htm.

Fisher, Robert, et al. CVonline: The Evolving, Distributed, Non-Proprietary, On-Line Compendium of Computer Vision. http://homepages.inf.ed.ac.uk/rbf/CVonline.

Huber, Daniel, et al. The Computer Vision Homepage. http://www-2.cs.cmu.edu/~cil/vision.html.

Krueger, Myron. *Artificial Reality II*. Addison-Wesley Professional, 1991.

Levin, Golan and Lieberman, Zachary. *Messa di Voce*. Interactive installation, 2003. http://www.tmema.org/messa.

Levin, Golan, and Zachary Lieberman. "In-Situ Speech Visualization in Real-Time Interactive Installation and Performance." *Proceedings of the Third International Symposium on Non-Photorealistic Animation and Rendering*. Annecy, France, June 7-9, 2004. http://www.flong.com/writings/pdf/messa_NPAR_2004_150dpi.pdf.

Lozano-Hemmer, Rafael. *Standards and Double Standards*. Interactive installation. http://www.fundacion.telefonica.com/at/rlh/eproyecto.html.

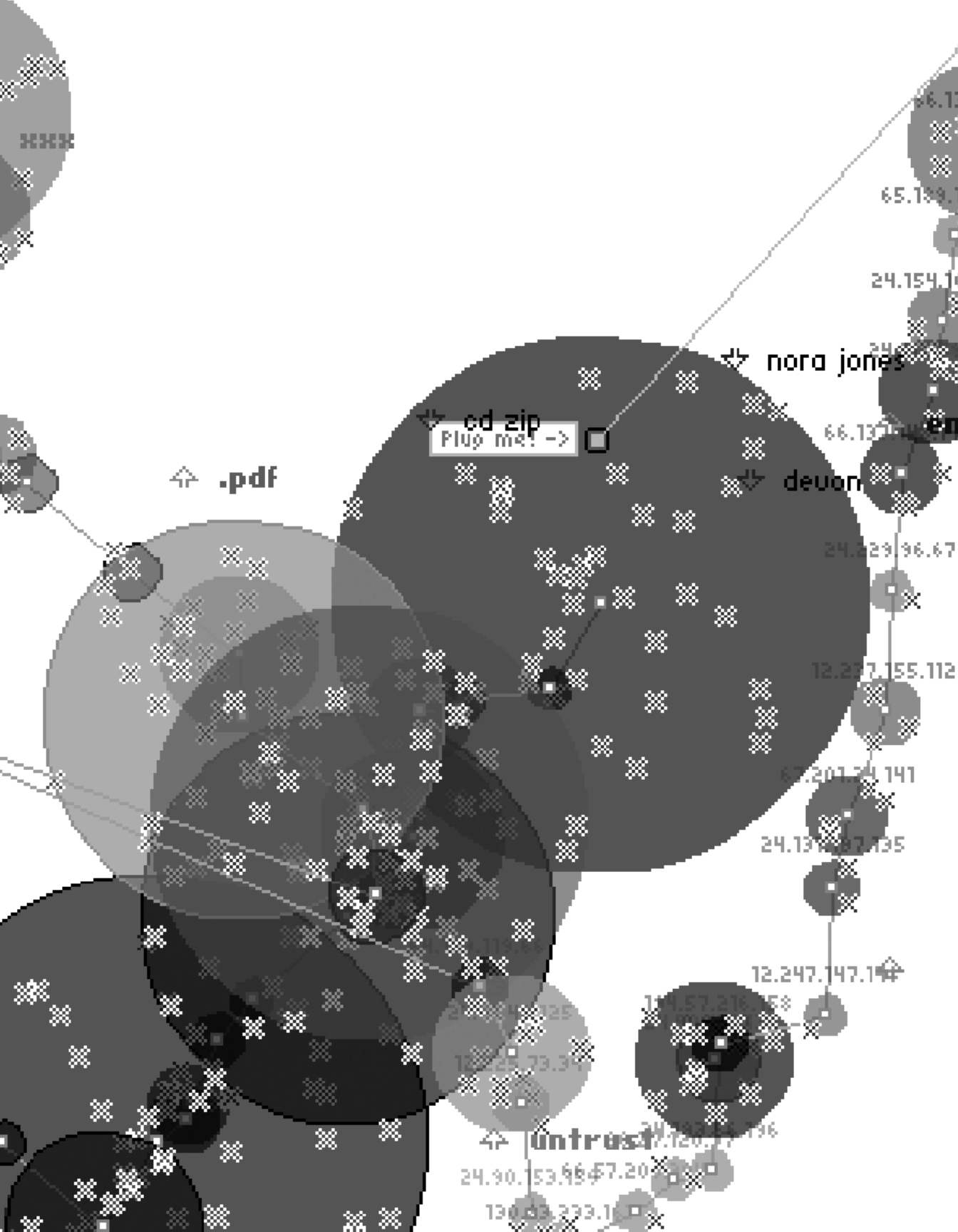
Melles Griot Corporation. Machine Vision Lens Fundamentals. http://www.mellesgriot.com/pdf/pg11-19.pdf.

Möller, Christian. *Cheese*. Installation artwork, 2003. http://www.christian-moeller.com.

Rokeby, David. *Sorting Daemon*. Computer-based installation, 2003. http://homepage.mac.com/davidrokeby/sorting.html.

Shachtman, Noah. "Tech and Art Mix at RNC Protest." *Wired News*, 27 August 2004. http://www.wired.com/news/culture/0,1284,64720,00.html.

Sparacino, Flavia. "(Some) computer vision based interfaces for interactive art and entertainment installations." INTER_FACE Body Boundaries, issue edited by Emanuele Quinz. *Anomalie* no. 2. Anomos, 2001. http://www.sensingplaces.com/papers/Flavia_isea2000.pdf.


nora jones
cd zip
Plug me! ->
.pdf
devon
untrust

Extension 4: Network

Text by Alexander R. Galloway

Networks are complex organizational forms. They bring into association discrete entities or nodes, allowing these nodes to connect to other nodes and indeed to other networks. Networks exist in the world in a vast variety of forms and in even more contexts: political, social, biological, and otherwise. While artists have used networks in many ways—from postal networks used to disseminate work to informal networks of artistic collaborators and larger aesthetic movements—this section looks specifically at a single instance of network technology, the Internet, and how artists have incorporated this technology into their work. There are two general trends: art making where the Internet is used as a tool for quick and easy dissemination of the work, and art making where the Internet is the actual medium of the work. These two trends are not mutually exclusive, however. Some of the most interesting online work weaves the two techniques together into exciting new forms that surpass the affordances of either technique.

The Internet and the arts

"In December 1995 Vuk Cosic got a message . . ." or so begins the tale of how "net.art," the niche art movement concerned with making art in and of the Internet, got its start and its name. As Alexei Shulgin explains in a posting to the Nettime Email list two years later, Cosic, a Slovenian artist, received an Email posted from an anonymous mailer. Apparently mangled in transit, the message was barely legible. "The only fragment of it that made any sense looked something like: [...] J8~g#|\;Net. Art{-^s1 [...]."[1]

Anonymous, accidental, glitchy, and slightly apocryphal—these are all distinctive characteristics of the net.art style, as seen in Web-based work from Cosic, Shulgin, Olia Lialina, Jodi, Heath Bunting, and many others. As Marina Grzinic writes, the "delays in transmission-time, busy signals from service providers, [and] crashing web browsers" contributed greatly to the way artists envisioned the aesthetic potential of the Web, a tendency that ran counter to the prevailing wisdom at the time of dot-com go, go, go.[2] Indeed many unsuspecting users assume that Jodi's Web-based and downloadable software projects have as their primary goal the immediate infection and ruin of one's personal computer. (Upon greater scrutiny it must be granted that this is only a secondary goal.) Perhaps peaking in 1998 with the absurd, anarchist experiments shoveled out on the 7-11 Email list—spoofs and shenanigans were par for the course due to the fact that the list administration tool, including subscriptions, header and footer variables, and moderation rules, was world read-writable by any Web surfer—the net.art movement is today viewable in captivity in such catchall publications as the hundred-contributor-strong anthology *Readme!*, edited in several cities simultaneously and

Schoenerwissen/OfCD. *Minitasking*, 2002. Image courtesy of Schoenerwissen/OfCD.

published by Autonomedia in 1999; the equally ecumenical anthology *NTNTNT* that emerged from CalArts in 2003; or Tilman Baumgärtel's two volumes of interviews, *net.art* (1999) and *net.art 2.0* (2001).

At the same time, buoyed by the dynamism of programming environments like Java and Flash, artists and designers began making online work that not only was "in and of" the Internet, but leveraged the net as a tool for quick and easy dissemination of executable code, both browser-based and otherwise. John Maeda created a number of sketches and games dating from the mid-1990s, including a series of interactive calendars using visual motifs borrowed from both nature and mathematics. Joshua Davis also emerged as an important figure through his online works *Praystation* and *Once-Upon-A-Forest*. Like Maeda, Davis fused algorithmic drawing techniques with an organic sense of composition.

It is worth recalling the profound sense of optimism and liberation that the Web brought to art and culture in the middle 1990s. All of a sudden tattooed skaters like Davis were being shuttled around the globe to speak to bespectacled venture capitalists about the possibilities of networking, generative code, and open software projects. And bespectacled philosophers like Geert Lovink were being shuttled around the globe to speak to tattooed skaters about—what else—the possibilities of networking, generative code, and open software projects. Everything was topsy-turvy. Even the net.art movement, which was in part influenced by Lovink's suspicion of all things "wired" and Californian, was nonetheless propelled by the utopian promise of networks, no matter that sometimes those networks had to be slashed and hacked in the process. Networks have, for several decades, acted as tonics for and inoculations against all things centralized and authoritarian, be they Paul Baran's 1964 schematics for routing around both the AT&T/Bell national telephone network and the then impending sorties of Soviet ICBMs; or the grassroots networks of the 1960s new social movements, which would later gain the status of art in Deleuze and Guattari's emblematic literary concept of the "rhizome," quite literally a grassroots model of networked being; or indeed the much earlier and oft-cited remarks from Bertolt Brecht on the early revolutionary potential of radio networks (reprised, famously, in Hans Magnus Enzensberger's 1974 essay on new media "Constituents for a Theory of the Media"). In other words, the arrival of the Web in the middle to late 1990s generated much excitement in both art and culture, for it seemed like a harbinger for the coming of some new and quite possibly revolutionary mode of social interaction. Or, as Cosic said once, with typical bravado, "all art to now has been merely a substitute for the Internet."

It is also helpful to contextualize these remarks through reference to the different software practices of various artists and movements. Each software environment is a distinct medium. Each grants particular aesthetic affordances to the artist and diminishes others. Each comes with a number of side effects that may be accentuated or avoided, given the proclivities of the artist. So, while acknowledging that digital artists' tools and materials tend to vary widely, it is perhaps helpful to observe that the net.art scene (Bunting, Shulgin, Lialina, Jodi, et al.), particularly during the period 1995–2000, coded primarily in browser-based markup languages such as HTML, with the addition of Javascript for the execution of basic algorithms. A stripped-down, "text only" format was

distinctive of this period. One gag used by a number of different artists was not to have a proper homepage at all, but instead to use Apache's default directory index of files and folders. The stripped-down approach did not always deliver simplicity to the user, however, as in the case of Jodi's early homepage (now archived at *http://wwwwwwww.jodi.org*) in which they neglected a crucial <pre> tag and then, in an apparent attempt to overcompensate for the first glitch, encircled the page in a <blink> tag no less prickly on the eyes as the missing <pre> tag is disorienting. The blinking page throbbed obscenely in the browser window, one glitch thus compounding the other. Created as an unauthorized addition to HTML by Netscape Navigator, the blink tag essentially vanished from the Internet as Internet Explorer became more dominant in the late 1990s. So today the Jodi page doesn't blink. One wonders which exactly is the work: the op-art, strobe effect that appeared in the Netscape browser window during the years when people used Netscape, or the HTML source still online today in which the work is "explained" to any sleuth willing to trace the narrative of markup tags missing and misplaced?

While artists had used fixed-width ASCII fonts and ANSI characters as design elements long before the popularization of the Web in the mid-1990s, it was the creation of HTML in 1993 (synchronized with its use in the newly invented Web servers and Web browsers like Netscape) that transformed the Internet into a space for the visual arts. HTML established itself quickly as the most influential mark-up language and graphic design protocol for two reasons: first, the text-only nature of HTML made it low-bandwidth-friendly during a time when most users connected via modems and phone lines; and second, HTML is a protocol, meaning that it acts as a common denominator (the <blink> tag notwithstanding) bridging a wide variety of dissimilar technical platforms. But, as seen in the work of Davis, which gravitates toward Flash but also includes Web, print, and video, one must not overemphasize HTML as an aesthetic medium. During this same period the network delivery of executable code (Java applets, Flash, Shockwave, and so on) also became more and more exciting as a medium for art-making, as did CUSeeMe, Web radio, video, and other streaming content that operates outside of the normal bounds of the browser frame. John Simon's 1997 *Every Icon* was written as a Java applet and therefore easily deliverable online as executable code. In what Lev Manovich has dubbed "Generation Flash," a whole new community sprang up, involving artists like Yugo Nakamura, Matt Owens, and James Paterson and intersecting with both dot-com startups like i|o 360° and Razorfish (or the artist's own design shops) and indie youth culture. Their medium is not solely the text-only markup codes of HTML but also the more sophisticated Macromedia languages (ActionScript and Lingo) as well as Javascript, Java, and server-side languages like Perl and PHP.

Internet protocols and concepts

In order to understand how online works are made and viewed, it will be useful to address a number of key concepts in the area of computer networking. A computer network consists of two or more machines connected via a data link. If a networked

computer acts primarily as a source of data, it is called a *server*. A server typically has a fixed address, is online continuously, and functions as a repository for files which are transmitted back to any other computers on the network that request them. If a networked computer acts primarily as a solicitor of information, it is called a *client*. For example, in checking one's Email, one acts as a client. Likewise, the machine where the Email is stored (the machine named after the @ sign in the Email address) acts as a server. These terms are flexible; a machine may act as a server in one context and as a client in another.

Any machine connected to the Internet, be it client or server, is obligated to have an address. On the Internet these addresses are called *IP addresses* and come in the form 123.45.67.89. (A new addressing standard is currently being rolled out that makes the addresses slightly longer.) Since IP addresses change from time to time and are difficult to remember, a system of natural-language shortcuts called the Domain Name System (DNS) allows IP addresses to be substituted by *domain names* such as "processing.org" or "google.com." In a Web address the word immediately preceding the domain name is the *host* name; for Web servers it is customary to name the host machine "www" after the World Wide Web. But this is only customary. In fact a Web server's host name can be most anything at all.

One of the main ways in which visual artists have used the Internet in their work is to conceive of the network as one giant database, an input stream that may be spidered, scanned, and parsed using automated clients. This is an artistic methodology that acknowledges the fundamental mutability of data (what programmers call "casting" a variable from one data type to another) and uses various data sources as input streams to power animations, to populate arrays of numbers with pseudo-random values, to track behavior, or quite simply for "content." Lisa Jevbratt's work *1:1* does this through the premise that every IP address might be represented by a single pixel. Her work scans the IP address namespace, number by number, pinging each address to determine whether a machine is online at that location. The results are visualized as pixels in a gigantic bitmap that, quite literally, represents the entire Internet (or at least all those machines with fixed IP addresses). In a very different way, Mark Napier's two works *Shredder* and *Digital Landfill* rely on a seemingly endless influx of online data, rearranging and overlaying source material in ways unintended by the original creators. Works like *Carnivore* (more on this below) and *Minitasking* approach the network itself as a data source, the former tapping into real-time Web traffic, and the latter tapping into real-time traffic on the Gnutella peer-to-peer network. Earlier works such as *I/O/D 4* (known as "The Webstalker"), or Jodi's *Wrongbrowser* series of alternative Web browsers also illustrate this approach, that the network itself is the art. All of these works automate the process of grabbing data from the Internet and manipulating it in some way. One of the most common types is a Web client, a piece of software that automates the process of requesting and receiving remote files on the World Wide Web.

Example 1: Web client (p. 572)
Processing's Net library includes ready-made classes for both servers and clients. In order to fetch a page from the Web, first one creates a client and connects to the address of the remote server. Using a simple call-and-response technique, the client requests the file, and the file is returned by the server. This call-and-response is defined by a protocol called Hypertext Transfer Protocol (HTTP). HTTP consists of a handful of simple commands that are used to describe the state of the server and client, to request files, and to post data back to the server if necessary. The most basic HTTP command is `GET`. This command is similar to filling out a book request form in a library: the client requests a file by name, the server "gets" that file and returns it to the client. HTTP also includes a number of response codes to indicate that the file was found successfully, or to indicate if any errors were encountered (for example, if the requested file doesn't exist). The command `GET / HTTP/1.0\n` means that the client is requesting the default file in the root web directory (`/`) and that the client is able to communicate using HTTP version 1.0. The trailing `\n` is the newline character, or roughly equivalent to hitting the return key. If the default file exists, the server transmits it back to the client.

While most computers have only a single Ethernet *port* (or wireless connection), the entire connectivity of each machine is able to sustain more connections than a single input or output, because the concept of a port is abstracted into software and the functionality of the port is thus duplicated many times over. In this way each networked computer is able to multitask its single network connection across scores of different connections (there are 1,024 well-known ports, and 65,535 ports in all). Thus ports allow a networked computer to communicate simultaneously on a large number of "channels" without blocking other channels or impeding the data flow of applications. For example, it is possible to read Email and surf the Web simultaneously because Email arrives through one port while Web pages use another. The union of IP address and port number (example: 123.45.67.89:80) is called a *socket*. Socket connections are the bread and butter of networking.

Example 2: Shared drawing canvas (p. 572)
Using the Processing Net library, it is possible to create a simple server. The example shows a server that shares a drawing canvas between two computers. In order to open a socket connection, a server must select a port on which to listen for incoming clients and through which to communicate. Although any port number may be used, it is best practice to avoid using port numbers already assigned to other network applications and protocols. Once the socket is established, a client may connect to the server and send or receive commands and data.

Paired with this server, the Processing `Client` class is instantiated by specifying a remote address and port number to which the socket connection should be made. Once the connection is made, the client may read (or write) data to the server. Because clients and servers are two sides of the same coin, the code examples are nearly identical for both. For this example, current and previous mouse coordinates are sent between client and server several times per second.

Example 3: Yahoo! Search SDK (p. 574)
As the Internet evolves from a relatively simple network of files and servers into a network where data is vended in more customized and focused ways, the capability of Web clients to target specific data sources on the net will become more and more prevalent. Consider the difference between surfing to a weather website to learn the current temperature, versus pinging a Web service with a ZIP code and having that server reply with a single number referring to the Fahrenheit temperature for that ZIP code. For the Web programmer, this evolution is welcome because it greatly simplifies the act of fetching and parsing online data by uncoupling those data from the sea of unnecessary HTML text that surrounds them on any typical Web page. One such Web service is the Yahoo! search engine. Using the Yahoo! Search SDK, it is possible to issue search queries programmatically. (This circumvents the former technique of using an HTTP client to post and receive search engine queries, which then must be stripped of HTML and parsed as variables.) The Yahoo! Search SDK essentially black-boxes the Web connection. This example uses the SDK to connect to the Yahoo! server and search for "processing.org." By default it returns the first twenty results, but that number can be adjusted. For each result, the web page title and its URL are printed to the console.

Example 4: Carnivore client (p. 575)
If a lower-level examination of the flows of data networks is desired, the Carnivore library for Processing allows the programmer to run a *packet sniffer* from within the Processing environment. A packet sniffer is any application that is able to indiscriminately eavesdrop on data traffic traveling through a local area network (LAN), even traffic not addressed to the machine running the sniffer. While this might sound unorthodox, and indeed a machine running a sniffer is described as being in "promiscuous mode," packet-sniffing technologies are as ubiquitous as the Internet itself and just as old. Systems administrators use packet sniffers to troubleshoot networking bugs. All Macintosh machines ship with the packet sniffer *tcpdump* preinstalled, while Windows and Linux users have an assortment of free sniffers (including tcpdump and its variants) to choose from. The Carnivore library for Processing merely simplifies the act of sniffing packets, making real-time traffic monitoring simple and easy to implement for any artist wishing to do so. Packets captured via Carnivore can be visualized in map form, parsed for keywords, or simply used for any type of algorithm that requires a steady stream of nonrandom event triggers.

Carnivore is a good stepping stone into the final area of computer networking discussed here, the Internet protocols. A protocol is a technological standard. The Internet protocols are a series of documents that describe how to implement standard Internet technologies such as data routing, handshaking between two machines, network addressing, and many other technologies. Two protocols have already been discussed—HTML, which is the language protocol for hypertext layout and design; and HTTP, which is the protocol for accessing Web-accessible files—but there are a few other protocols worth discussing in this context.

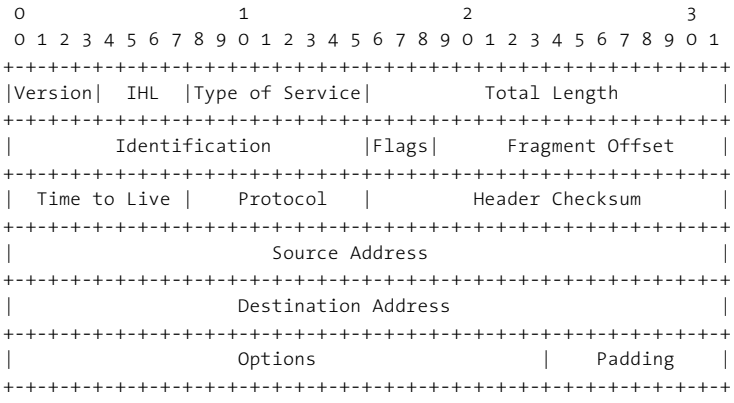
Protocols are abstract concepts, but they are also quite material and manifest themselves in the form of structured data headers that prepend and encapsulate all content traveling through the Internet. For example, in order for a typical HTML page to travel from server to client, the page is prepended by an HTTP header (a few lines of text similar to the `GET` command referenced previously). This glob of data is itself prepended by two additional headers, first a Transmission Control Protocol (TCP) header and next by an Internet Protocol (IP) header. Upon arrival at its destination, the message is unwrapped: the IP header is removed, followed by the TCP header, and finally the HTTP header is removed to reveal the original HTML page. All of this is done in the blink of an eye. All headers contain useful information about the packet. But perhaps the four most useful pieces of information are the sender IP address, receiver IP address, sender port, and receiver port. These four pieces are significant because they indicate the network addresses of the machines in question, plus, via a reverse lookup of the port numbers, the type of data being transferred (port 80 indicating Web data, port 23 indicating a Telnet connection, and so on). See the */etc/services* file on any Macintosh, Linux, or UNIX machine, or browse IANA's registry for a complete listing of port numbers. The addresses are contained in the IP header from byte 12 to byte 29 (counting from 0), while the ports are contained in bytes zero through three of the TCP header.

The two elements of the socket connection (IP address and port) are separated into two different protocols because of the different nature of IP and TCP. The IP protocol is concerned with routing data from one place to another and hence requires having an IP address in order to route correctly but cares little about the type of data in its payload. TCP is concerned with establishing a virtual circuit between server and client and thus requires slightly more information about the type of Internet communication being attempted. IP and TCP work so closely together that they are often described in one breath as the "TCP/IP suite."

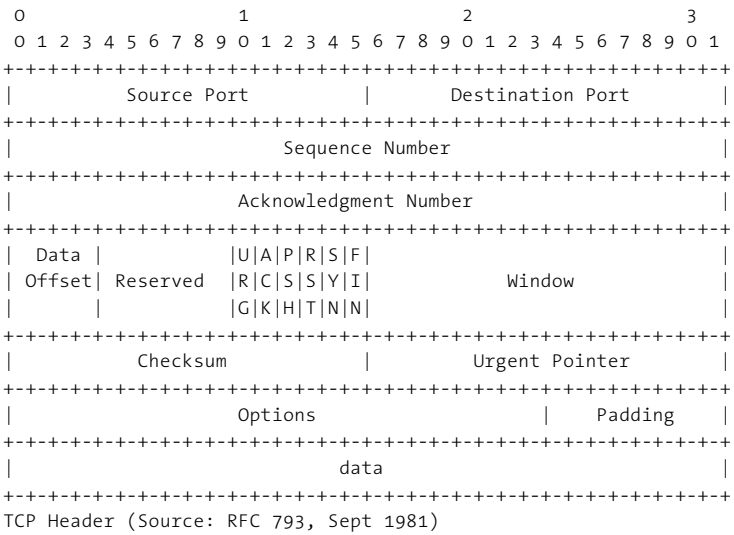
While most data on the Internet relies on the TCP/IP suite to get around, certain forms of networked communication are better suited to the UDP/IP combination. User Datagram Protocol (UDP) has a much leaner implementation than TCP, and while it therefore sacrifices many of the features of TCP it is nevertheless useful for stateless data connections and connections that require a high throughput of packets per second, such as online games.

Network tools

There are a number of existing network tools that a programmer may use beyond the Processing environment. Carnivore and tcpdump, two different types of packet sniffers that allow one to receive LAN packets in real time, have already been mentioned. The process of scanning networks for available hosts, called network discovery, is also possible using port scanner tools such as Nmap. These tools use a variety of methods for looping through a numerical set of IP addresses (example: 192.168.1.x where x is incremented from 0 to 255), testing to see if a machine responds at that address. Then if a machine is known to be online, the port scanner is used to loop through a range of ports



Internet Datagram Header (Source: RFC 791, Sept 1981)



TCP Header (Source: RFC 793, Sept 1981)

Headers

These diagrams specify how information in the IP and TCP headers are organized. IP addresses for sender and receiver are indicated, as well as other vital information such as the packet's "time to live" and checksums to ensure the integrity of the payload.

on the machine (example: 192.168.1.1:x where x is a port number incremented from 1 to 1024) in order to determine which ports are open, thereby determining which application services are available. Port scans can also be used to obtain "fingerprints" for remote machines, which aid in the identification of the machine's current operating system and type and version information for known application services.

Perhaps the most significant advance in popular networking since the emergence of the Web in the mid-1990s was the development of the Gnutella protocol in 2000. Coming on the heels of Napster, Gnutella fully distributed the process of file sharing and transfer, but also fully distributed the network's search algorithm, a detail that had created bottlenecks (not to mention legal liabilities) for the more centralized Napster. With a distributed search algorithm, search queries hopscotch from node to node, just like the "hot potato" method used in IP routing; they do not pass through any centralized server. The Gnutella protocol has been implemented in dozens of peer-to-peer applications. Several open source Gnutella "cores" are also available for use by developers, including the Java core for Limewire, which with a little ingenuity could easily be linked to Processing.

More recently, Bittorent, a peer-to-peer application that allows file transfers to happen simultaneously between large numbers of users, has been in wide use, particularly for transfers of large files such as video and software.

Many software projects requiring networked audio have come to rely on the Open Sound Control (OSC) protocol. OSC is a protocol for communication between multimedia devices such as computers and synthesizers. OSC has been integrated into SuperCollider and Max/MSP and has been ported to most modern languages including Perl and Java. Andreas Schlegel's "oscP5" is an OSC extension library for Processing.

Conclusion

Programmers are often required to consider interconnections between webs of objects and events. Because of this, programming for networks is a natural extension of programming for a single machine. Classes send messages to other classes just like hosts send messages to other hosts. An object has an interface, and so does an Ethernet adapter. The algorithmic construction of entities in dialog—pixels, bits, frames, nodes—is central to what Processing is all about. Networking these entities by moving some of them to one machine and some to another is but a small additional step. What is required, however, is a knowledge of the various standards and techniques at play when bona fide networking takes place.

Historically, there have been two basic strands of networked art: art where the network is used as the actual medium of art-making, or art where the network is used as the transportation medium for dissemination of the work. The former might be understood as art *of* the Internet, while the latter as art *for* the Internet. The goal of this text has been to introduce some of the basic conditions, both technological and aesthetic, for making networked art, in the hopes that entirely new techniques and approaches will spring forth in the future as both strands blend together into exciting new forms.

Notes

1. Alexei Shulgin, "Net.Art - the origin," Nettime mailing list archives, 18 March 1997, http://nettime.org/Lists-Archives/nettime-l-9703/msg00094.html.
2. Marina Grzinic, "Exposure Time, the Aura, and Telerobotics," in *The Robot in the Garden: Telerobotics and Telepistemology in the Age of the Internet*, edited by Ken Goldberg (MIT Press, 2000), p. 215.

Code

Example 1: Web client

```
// A simple Web client using HTTP

import processing.net.*;

Client c;
String data;

void setup() {
  size(200, 200);
  background(50);
  fill(200);
  c = new Client(this, "www.processing.org", 80);  // Connect to server on port 80
  c.write("GET / HTTP/1.0\n");  // Use the HTTP "GET" command to ask for a Web page
  c.write("Host: my_domain_name.com\n\n"); // Be polite and say who we are
}

void draw() {
  if (c.available() > 0) {    // If there's incoming data from the client...
    data += c.readString();   // ...then grab it and print it
    println(data);
  }
}
```

Example 2A: Shared drawing canvas (server)

```
import processing.net.*;

Server s;
Client c;
String input;
int data[];

void setup() {
  size(450, 255);
  background(204);
  stroke(0);
  frameRate(5); // Slow it down a little
  s = new Server(this, 12345);  // Start a simple server on a port
}
```

```
void draw() {
  if (mousePressed == true) {
    // Draw our line
    stroke(255);
    line(pmouseX, pmouseY, mouseX, mouseY);
    // Send mouse coords to other person
    s.write(pmouseX + " " + pmouseY + " " + mouseX + " " + mouseY + "\n");
  }

  // Receive data from client
  c = s.available();
  if (c != null) {
    input = c.readString();
    input = input.substring(0, input.indexOf("\n"));  // Only up to the newline
    data = int(split(input, ' '));  // Split values into an array
    // Draw line using received coords
    stroke(0);
    line(data[0], data[1], data[2], data[3]);
  }
}
```

Example 2B: Shared drawing canvas (client)

```
import processing.net.*;
Client c;
String input;
int data[];

void setup() {
  size(450, 255);
  background(204);
  stroke(0);
  frameRate(5); // Slow it down a little
  // Connect to the server's IP address and port
  c = new Client(this, "127.0.0.1", 12345); // Replace with your server's IP and port
}

void draw() {
  if (mousePressed == true) {
    // Draw our line
    stroke(255);
    line(pmouseX, pmouseY, mouseX, mouseY);
    // Send mouse coords to other person
    c.write(pmouseX + " " + pmouseY + " " + mouseX + " " + mouseY + "\n");
  }

  // Receive data from server
  if (c.available() > 0) {
    input = c.readString();
    input = input.substring(0,input.indexOf("\n"));  // Only up to the newline
    data = int(split(input, ' '));  // Split values into an array
    // Draw line using received coords
```

```
    stroke(0);
    line(data[0], data[1], data[2], data[3]);
  }
}
```

Example 3: Yahoo! API

```
// Download the Yahoo! Search SDK from http://developer.yahoo.com/download
// Inside the download, find the yahoo_search-2.X.X.jar file somewhere inside
// the "Java" subdirectory. Drag the jar file to your sketch and it will be
// added to your 'code' folder for use.
// This example is based on the Yahoo! API example

// Replace this with a developer key from http://developer.yahoo.com
String appid = "YOUR_DEVELOPER_KEY_HERE";
SearchClient client = new SearchClient(appid);

String query = "processing.org";
WebSearchRequest request = new WebSearchRequest(query);
// (Optional) Set the maximum number of results to download
//request.setResults(30);

try {
  WebSearchResults results = client.webSearch(request);

  // Print out how many hits were found
  println("Displaying " + results.getTotalResultsReturned() +
          " out of " + results.getTotalResultsAvailable() + " hits.");
  println();

  // Get a list of the search results
  WebSearchResult[] resultList = results.listResults();

  // Loop through the results and print them to the console
  for (int i = 0; i < resultList.length; i++) {
    // Print out the document title and URL.
    println((i + 1) + ".");
    println(resultList[i].getTitle());
    println(resultList[i].getUrl());
    println();
  }

// Error handling below; see the documentation of the Yahoo! API for details

} catch (IOException e) {
  println("Error calling Yahoo! Search Service: " + e.toString());
  e.printStackTrace();
} catch (SearchException e) {
  println("Error calling Yahoo! Search Service: " + e.toString());
  e.printStackTrace();
}
```

Example 4: Carnivore client

```
// Note: requires Carnivore Library for Processing v2.2 (http://r-s-g.org/carnivore)
// Windows, first install winpcap (http://winpcap.org)
// Mac, first open a Terminal and execute this commmand: sudo chmod 777 /dev/bpf*
// (must be done each time you reboot your Mac)

import java.util.Iterator;
import org.rsg.carnivore.*;
import org.rsg.carnivore.net.*;
HashMap nodes = new HashMap();
float startDiameter = 100.0;
float shrinkSpeed = 0.97;
int splitter, x, y;
PFont font;

void setup(){
  size(800, 600);
  background(255);
  frameRate(10);

  Log.setDebug(true); // Uncomment this for verbose mode
  CarnivoreP5 c = new CarnivoreP5(this);
  //c.setVolumeLimit(4);

  // Use the "Create Font" tool to add a 12 point font to your sketch,
  // then use its name as the parameter to loadFont().
  font = loadFont("CourierNew-12.vlw");
  textFont(font);
}

void draw() {
  background(255);
  drawNodes();
}

// Iterate through each node
synchronized void drawNodes() {
  Iterator it = nodes.keySet().iterator();
  while (it.hasNext()){
    String ip = (String)it.next();
    float d = float(nodes.get(ip).toString());

    // Use last two IP address bytes for x/y coords
    splitter = ip.lastIndexOf(".");
    y = int(ip.substring(splitter+1)) * height / 255; // Scale to applet size
    String tmp = ip.substring(0,splitter);
    splitter = tmp.lastIndexOf(".");
    x = int(tmp.substring(splitter+1)) * width / 255; // Scale to applet size

    // Draw the node
    stroke(0);
    fill(color(100, 200));  // Rim
```

```
    ellipse(x, y, d, d);    // Node circle
    noStroke();
    fill(color(100, 50));   // Halo
    ellipse(x, y, d + 20, d + 20);

    // Draw the text
    fill(0);
    text(ip, x, y);

    // Shrink the nodes a little
    nodes.put(ip, str(d * shrinkSpeed));
  }
}

// Called each time a new packet arrives
synchronized void packetEvent(CarnivorePacket packet){
  println("[PDE] packetEvent: " + packet);

  // Remember these nodes in our hash map
  nodes.put(packet.receiverAddress.toString(), str(startDiameter));
  nodes.put(packet.senderAddress.toString(), str(startDiameter));
}
```

Resources

Network software toolkits

Fenner, Bill, et al. Tcpdump. Packet sniffer, 1991. http://www.tcpdump.org.

Frankel, Justin, Tom Pepper, et al. Gnutella. Peer-to-peer network protocol, 2000. http://rfc-gnutella.sourceforge.net.

Google. Google Web APIs. Web service, 2004. http://www.google.com/apis.

Saito, Tatsuya. Google API Processing library, Extension library for Processing, 2005. http://www.processing.org/reference/libraries.

Internet Engineering Task Force (IETF). Standards-making organization. http://www.ietf.org.

Fyodor. Nmap. Port scanner, 1997. http://www.insecure.org/nmap.

RSG. Carnivore. Data monitoring toolkit, 2001. http://r-s-g.org/carnivore.

RSG. Carnivore Library for Processing. Extension library for Processing, 2005. http://r-s-g.org/carnivore/processing.php.

World Wide Web Consortium (W3C). Standards-making organization. http://www.w3.org.

Wright, Matt, et al. Open Sound Control (OSC). Audio networking protocol, 1997. http://www.cnmat.berkeley.edu/OpenSoundControl.

Schlegel, Andreas. oscP5. OSC extension library for Processing, 2004. http://www.sojamo.de/iv/index.php?n=11.

Texts and artworks

Baumgärtel, Tilman, ed. *Net.art 2.0: New Materials towards Net Art*. Verlag Für Moderne Kunst Nürnberg, 2001.

Baumgärtel, Tilman, ed. *Net.art: Materialien zur Netzkunst*. Verlag Für Moderne Kunst Nürnberg, 1999.

Bosma, Josephine, et al., eds., *Readme!* Autonomedia, 1999.

Brown, Jason, ed., *NTNTNT*. CalArts School of Art, 2003.

Davis, Joshua. *Praystation.com*. Website, 1999. http://praystation.com.

Davis, Joshua. *Praystation Harddrive* (CD-ROM). Systems Design Limited, 2001.

Hall, Eric. *Internet Core Protocols: The Definitive Guide*. O'Reilly, 2000.

Escape. I/O/D 4: "The Web Stalker." Software application, 1997. http://www.backspace.org/iod.

Grzinic, Marina. "Exposure Time, the Aura, and Telerobotics," in *The Robot in the Garden: Telerobotics and Telepistemology in the Age of the Internet*, edited by Ken Goldberg. MIT Press, 2000.

Internet Assigned Numbers Authority (IANA). List of port numbers. http://www.iana.org/assignments/port-numbers.

Jodi. %Location | http://wwwwwwwww.jodi.org. Website, 1995. http://wwwwwwwww.jodi.org.

Jodi. *Wrongbrowser*. Software application, 2001. http://www.wrongbrowser.com.

Jevbratt, Lisa. *1:1*. Website, 1999. http://www.c5corp.com/1to1.

Nakamura, Yugo. *Yugop.com*. Website, 1999. http://yugop.com.

Napier, Mark. *The Digital Landfill*. Website, 1998. http://www.potatoland.org/landfill.

Napier, Mark. *Shredder*. Website, 1998. http://www.potatoland.org/shredder.

Owens, Matt. *Volumeone*. Website and design studio, 1997. http://www.volumeone.com.

Paterson, James. *Presstube*. Website, 2002. http://www.presstube.com.

Postel, Jonathan, et al. "Internet Protocol." RFC 791, September 1981.

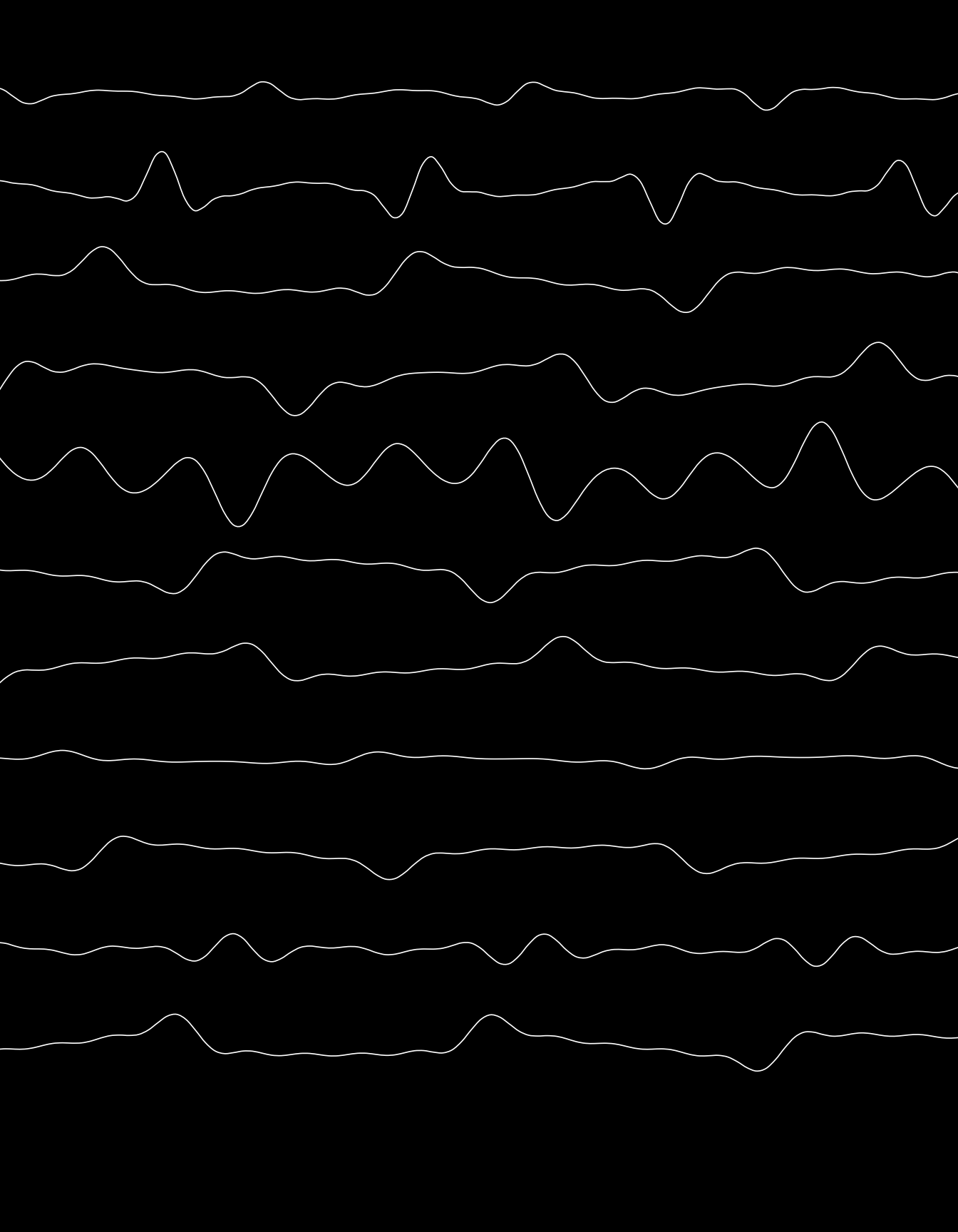
Postel, Jonathan, et al. "Transmission Control Protocol." RFC 793, September 1981.

Shulgin, Alexei. "Net.Art – the origin." Netttime mailing list archives, 18 March 1997. http://amsterdam.nettime.org/Lists-Archives/nettime-l-9703/msg00094.html.

Simon, John F. Jr. *Every Icon*. Java applet, 1997. http://www.numeral.com/everyicon.html.

Schoenerwissen/OfCD. *Minitasking*. Software application, 2002. http://minitasking.com.

Stevens, W. Richard. *TCP/IP Illustrated*. Volume 1, The Protocols. Addison-Wesley, 1994.



Extension 5: Sound

Text by R. Luke DuBois

The history of music is, in many ways, the history of technology. From developments in the writing and transcription of music (notation) to the design of spaces for the performance of music (acoustics) to the creation of musical instruments, composers and musicians have availed themselves of advances in human understanding to perfect and advance their professions. Unsurprisingly, therefore, we find that in the machine age these same people found themselves first in line to take advantage of the new techniques and possibilities offered by electricity, telecommunications, and, in the last century, digital computers to leverage all of these systems to create new and expressive forms of sonic art. Indeed, the development of phonography (the ability to reproduce sound mechanically) has, by itself, had such a transformative effect on aural culture that it seems inconceivable now to step back to an age where sound could emanate only from its original source.[1] The ability to create, manipulate, and losslessly reproduce sound by digital means is having, at the time of this writing, an equally revolutionary effect on how we listen. As a result, the artist today working with sound has not only a huge array of tools to work with, but also a medium exceptionally well suited to technological experimentation.

Music and sound programming in the arts

Thomas Edison's 1857 invention of the phonograph and Nikola Tesla's wireless radio demonstration of 1893 paved the way for what was to be a century of innovation in the electromechanical transmission and reproduction of sound. Emile Berliner's gramophone record (1887) and the advent of AM radio broadcasting under Guglielmo Marconi (1922) democratized and popularized the consumption of music, initiating a process by which popular music quickly transformed from an art of minstrelsy into a commodified industry worth tens of billions of dollars worldwide.[2] New electronic musical instruments, from the large and impractical telharmonium to the simple and elegant theremin multiplied in tandem with recording and broadcast technologies and prefigured the synthesizers, sequencers, and samplers of today. Many composers of the time were, not unreasonably, entranced by the potential of these new mediums of transcription, transmission, and performance. Luigi Russolo, the futurist composer, wrote in his 1913 manifesto The *Art of Noises* of a futurist orchestra harnessing the power of mechanical noisemaking (and phonographic reproduction) to "liberate" sound from the tyranny of the merely musical. John Cage, in his 1937 monograph *Credo: The Future of Music*, wrote this elliptical doctrine:

> The use of noise to make music will continue and increase until we reach a music produced through the aid of electrical instruments which will make available for musical purposes any and all sounds that can be heard. Photoelectric, film, and mechanical mediums for the synthetic production of music will be explored. Whereas, in the past, the point of disagreement has been between dissonance and consonance, it will be, in the immediate future, between noise and so-called musical sounds.[3]

The invention and wide adoption of magnetic tape as a medium for the recording of audio signals provided a breakthrough for composers waiting to compose purely with *sound*. In the early postwar period, the first electronic music studios flourished at radio stations in Paris (ORTF) and Cologne (WDR). The composers at the Paris studio, most notably Pierre Henry and Pierre Schaeffer, developed the early compositional technique of *musique concrète*, working directly with recordings of sound on phonographs and magnetic tape to construct compositions through a process akin to what we would now recognize as sampling. Schaeffer's *Étude aux chemins de fer* (1948) and Henry and Schaeffer's *Symphonie pour un homme seul* are classics of the genre. Meanwhile, in Cologne, composers such as Herbert Eimart and Karlheinz Stockhausen were investigating the use of electromechanical oscillators to produce pure sound waves that could be mixed and sequenced with a high degree of precision. This classic *elektronische music* was closely tied to the serial techniques of the contemporary modernist avant-garde, who were particularly well suited aesthetically to become crucial advocates for the formal quantification and automation offered by electronic and, later, computer music.[4] The Columbia-Princeton Electronic Music Center, founded by Vladimir Ussachevsky, Otto Luening, Milton Babbitt, and Roger Sessions in New York in 1957, staked its reputation on the massive RCA Mark II Sound Synthesizer, a room-sized machine capable of producing and sequencing electronically generated tones with an unprecedented degree of precision and control. In the realm of popular music, pioneering steps were taken in the field of recording engineering, such as the invention of multitrack tape recording by the guitarist Les Paul in 1954. This technology, enabling a single performer to "overdub" her/himself onto multiple individual "tracks" that could later be mixed into a composite, filled a crucial gap in the technology of recording and would empower the incredible boom in recording-studio experimentation that permanently cemented the commercial viability of the studio recording in popular music.

Composers adopted digital computers slowly as a creative tool because of their initial lack of real-time responsiveness and intuitive interface. Although the first documented use of the computer to make music occurred in 1951 on the CSIRAC machine in Sydney, Australia, the genesis of most foundational technology in computer music as we know it today came when Max Mathews, a researcher at Bell Labs in the United States, developed a piece of software for the IBM 704 mainframe called MUSIC. In 1957, the MUSIC program rendered a 17-second composition by Newmann Guttmann called "In the Silver Scale". Originally tasked with the development of human-comprehensible synthesized speech, Mathews developed a system for encoding and decoding sound waves digitally, as well as a system for designing and implementing digital audio processes computationally. His assumptions about these representational schemes are still largely in use and will be described later in this text. The advent of faster machines,

computer music programming languages, and digital systems capable of real-time interactivity brought about a rapid transition from analog to computer technology for the creation and manipulation of sound, a process that by the 1990s was largely comprehensive.[5]

Sound programmers (composers, sound artists, etc.) use computers for a variety of tasks in the creative process. Many artists use the computer as a tool for the algorithmic and computer-assisted composition of music that is then realized off-line. For Lejaren Hiller's *Illiac Suite* for string quartet (1957), the composer ran an algorithm on the computer to generate notated instructions for live musicians to read and perform, much like any other piece of notated music. This computational approach to composition dovetails nicely with the aesthetic trends of twentieth-century musical modernism, including the controversial notion of the composer as "researcher," best articulated by serialists such as Milton Babbitt and Pierre Boulez, the founder of IRCAM. This use of the computer to manipulate the symbolic language of music has proven indispensable to many artists, some of whom have successfully adopted techniques from computational research in artificial intelligence to attempt the modeling of preexisting musical styles and forms; for example, David Cope's *5000 works…* and Brad Garton's *Rough Raga Riffs* use stochastic techniques from information theory such as Markov chains to simulate the music of J. S. Bach and the styles of Indian Carnatic sitar music, respectively.

If music can be thought of as a set of informatics to describe an organization of sound, the synthesis and manipulation of sound itself is the second category in which artists can exploit the power of computational systems. The use of the computer as a producer of synthesized sound liberates the artist from preconceived notions of instrumental capabilities and allows her/him to focus directly on the *timbre* of the sonic artifact, leading to the trope that computers allow us to make any sound we can imagine. Composers such as Jean-Claude Risset (*The Bell Labs Catalogue*), Iannis Xenakis (*GENDYN3*), and Barry Truax (*Riverrun*), have seen the computer as a crucial tool in investigating sound itself for compositional possibilities, be they imitative of real instruments (Risset), or formal studies in the stochastic arrangements of synthesized sound masses (Xenakis) using techniques culminating in the principles of granular synthesis (Truax). The computer also offers extensive possibilities for the assembly and manipulation of preexisting sound along the musique concrète model, though with all the alternatives a digital computer can offer. The compositional process of digital sampling, whether used in pop recordings (Brian Eno and David Byrne's *My Life in the Bush of Ghosts*, Public Enemy's *Fear of a Black Planet*) or conceptual compositions (John Oswald's *Plunderphonics*, Chris Bailey's *Ow, My Head*), is aided tremendously by the digital form sound can now take. Computers also enable the transcoding of an audio signal into representations that allow for radical reinvestigation, as in the time-stretching works of Leif Inge (*9 Beet Stretch*, a 24-hour "stretching" of Beethoven's Ninth Symphony) and the time-lapse phonography of this text's author (*Messiah*, a 5-minute "compression" of Handel's *Messiah*).

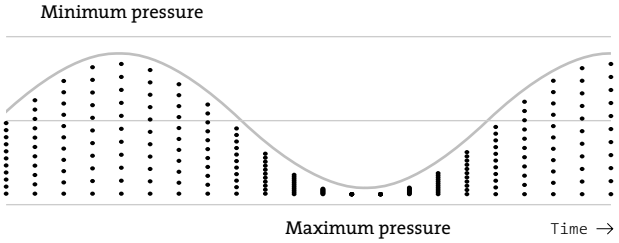
Artists working with sound will often combine the two approaches, allowing for the creation of generative works of sound art where the underlying structural system, as well as the sound generation and delivery, are computationally determined. Artists such

as Michael Schumacher, Stephen Vitiello, Carl Stone, and Richard James (the Aphex Twin) all use this approach. Most excitingly, computers offer immense possibilities as actors and interactive agents in sonic *performance*, allowing performers to integrate algorithmic accompaniment (George Lewis), hyperinstrument design (Laetitia Sonami, *Interface*), and digital effects processing (Pauline Oliveros, Mari Kimura) into their repertoire.

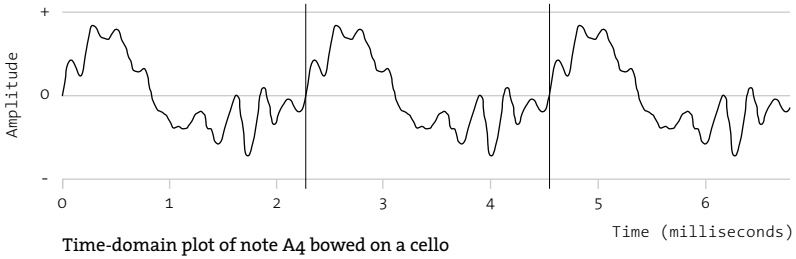
Now that we've talked a bit about the potential for sonic arts on the computer, we'll investigate some of the specific underlying technologies that enable us to work with sound in the digital domain.

Sound and musical informatics

Simply put, we define sound as a vibration traveling through a medium (typically air) that we can perceive through our sense of hearing. Sound propagates as a longitudinal wave that alternately compresses and decompresses the molecules in the matter (e.g., air) through which it travels. As a result, we typically represent sound as a plot of pressure over time:



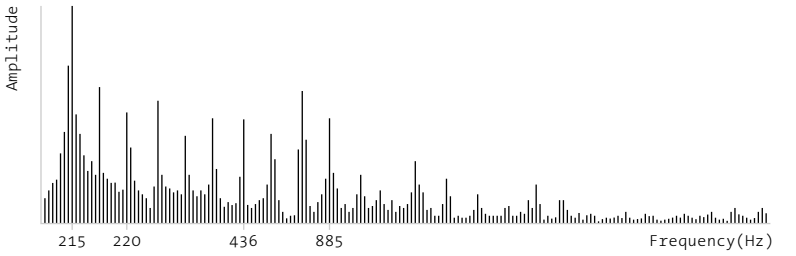
This time-domain representation of sound provides an accurate portrayal of how sound works in the real world, and, as we shall see shortly, it is the most common representation of sound used in work with digitized audio. When we attempt a technical description of a sound wave, we can easily derive a few metrics to help us better understand what's going on. In the first instance, by looking at the amount of displacement caused by the sound pressure wave, we can measure the amplitude of the sound. This can be measured on a scientific scale in *pascals* of pressure, but it is more typically quantified along a logarithmic scale of *decibels*. If the sound pressure wave repeats in a regular or *periodic* pattern, we can look at the wavelength of a single iteration of that pattern and from there derive the frequency of that wave. For example, if a sound traveling in a medium at 343 meters per second (the speed of sound in air at room temperature) contains a wave that repeats every half-meter, that sound has a frequency of 686 hertz, or cycles per second. The figure below shows a plot of a cello note sounding at 440 Hz; as a result, the periodic pattern of the waveform (demarcated with vertical lines) repeats every 2.27 ms:



Time-domain plot of note A4 bowed on a cello

Typically, sounds occurring in the natural world contain many discrete frequency components. In noisy sounds, these frequencies may be completely unrelated to one another or grouped by a typology of boundaries (e.g., a snare drum may produce frequencies randomly spread between 200 and 800 hertz). In *harmonic* sounds, however, these frequencies are often spaced in integer ratios, such that a cello playing a note at 200 hertz will produce frequencies not only at the *fundamental* of 200, but at multiples of 200 up the *harmonic series*, i.e., at 400, 800, 1200, 1600, 2000, and so on. A male singer producing the same note will have the same frequency components in his voice, though in different proportions to the cello. The presence, absence, and relative strength of these harmonics (also called *partials* or *overtones*) provide what we perceive as the timbre of a sound.

When a sound reaches our ears, an important sensory translation happens that is important to understand when working with audio. Just as light of different wavelengths and brightness excites different retinal receptors in your eyes to produce a color image, the cochlea of your inner ear contains an array of hair cells on the basilar membrane that are tuned to respond to different frequencies of sound. The inner ear contains hair cells that respond to frequencies spaced roughly between 20 and 20,000 hertz, though many of these hairs will gradually become desensitized with age or exposure to loud noise. These cells in turn send electrical signals via your auditory nerve into the auditory cortex of your brain, where they are parsed to create a frequency-domain image of the sound arriving in your ears:



Frequency-domain plot of a sustained note (220 Hz) bowed on a cello

This representation of sound, as a discrete "frame" of frequencies and amplitudes independent of time, is more akin to the way in which we perceive our sonic environment than the raw pressure wave of the time domain. Jean-Baptiste-Joseph

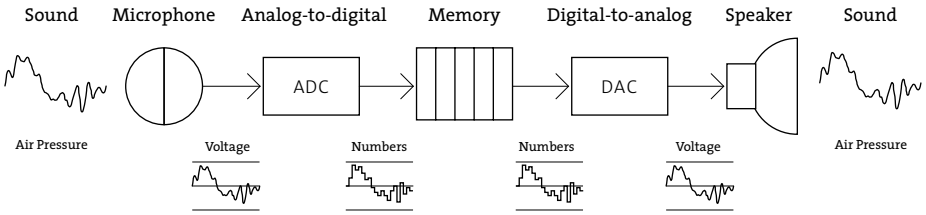
Fourier, a nineteenth-century French mathematician, developed the equations that allow us to translate a sound pressure wave (no matter how complex) into its constituent frequencies and amplitudes. This *Fourier transform* is an important tool in working with sound in the computer.

Our auditory system takes these streams of frequency and amplitude information from our two ears and uses them to construct an auditory "scene," akin to the visual scene derived from the light reaching our retinas.[6] Our brain analyzes the acoustic information based on a number of parameters such as onset time, stereo correlation, harmonic ratio, and complexity to parse out a number of acoustic sources that are then placed in a three-dimensional image representing what we hear. Many of the parameters that psychoacousticians believe we use to comprehend our sonic environment are similar to the grouping principles defined in Gestalt psychology.

If we loosely define music as the organization and performance of sound, a new set of metrics reveals itself. While a comprehensive overview of music theory, Western or otherwise, is well beyond the scope of this text, it's worth noting that there is a vocabulary for the description of music, akin to how we describe sound. Our system for perceiving *loudness* and *pitch* (useful "musical" equivalents to amplitude and frequency) work along a logarithmic scale, such that a tone at 100 hertz and a tone at 200 hertz are considered to be the same *distance* apart in terms of pitch as tones at 2000 and 4000 hertz. The distance between two sounds of doubling frequency is called the *octave*, and is a foundational principle upon which most culturally evolved theories of music rely. Most musical cultures then subdivide the octave into a set of pitches (e.g., 12 in the Western chromatic scale, 7 in the Indonesian *pelog* scale) that are then used in various collections (*modes* or *keys*). These pitches typically reflect some system of *temperament* or *tuning*, so that multiple musicians can play together; for example, the note A4 (the A above middle C) on the Western scale is usually calibrated to sound at 440 hertz in contemporary music.

Digital representation of sound and music

Sound typically enters the computer from the outside world (and vice versa) according to the time-domain representation explained earlier. Before it is digitized, the acoustic pressure wave of sound is first converted into an electromagnetic wave of sound that is a direct analog of the acoustic wave. This electrical signal is then fed to a piece of computer hardware called an analog-to-digital converter (ADC or A/D), which then digitizes the sound by sampling the amplitude of the pressure wave at a regular interval and quantifying the pressure readings numerically, passing them upstream in small packets, or vectors, to the main processor, where they can be stored or processed. Similarly, vectors of digital samples can be sent downstream from the computer to a hardware device called a digital-to-analog converter (DAC or D/A), which takes the numeric values and uses them to construct a smoothed-out electromagnetic pressure wave that can then be fed to a speaker or other device for playback:



Most contemporary digital audio systems (soundcards, etc.) contain both A/D and D/A converters (often more than one of each, for stereo or multichannel sound recording and playback) and can use both simultaneously (so-called full duplex audio). The specific system of encoding and decoding audio using this methodology is called PCM (or pulse-code modulation); developed in 1937 by Alec Reeves, it is by far the most prevalent scheme in use today.

The speed at which audio signals are digitized is referred to as the *sampling rate*; it is the resolution that determines the highest frequency of sound that can be measured (equal to half the sampling rate, according to the *Nyquist theorem*). The numeric resolution of each sample in terms of computer storage space is called the *bit depth*; this value determines how many discrete levels of amplitude can be described by the digitized signal. The digital audio on a compact disc, for example, is digitized at 44,100 hertz with a 16-bit resolution, allowing for frequencies up to 22,050 hertz (i.e., just above the range of human hearing) with 65,536 (216) different levels of amplitude possible for each sample. Professional audio systems will go higher (96 or 192 kHz at 24- or 32-bit resolution) while industry telephony systems will go lower (e.g., 8,192 Hz at 8-bit). Digitized sound representing multiple acoustic sources (e.g., instruments) or destinations (e.g., speakers) is referred to as multi-channel audio. Monaural sound consists of, naturally, only one stream; stereo (two-stream) audio is standard on all contemporary computer audio hardware, and various types of surround-sound (five or seven streams of audio with one or two special channels for low frequencies) are becoming more and more common.

Once in the computer, sound is stored using a variety of formats, both as sequences of PCM samples and in other representations. The two most common PCM sound file formats are the Audio Interchange File Format (AIFF) developed by Apple Computer and Electronic Arts and the WAV file format developed by Microsoft and IBM. Both formats are effectively equivalent in terms of quality and interoperability, and both are inherently *lossless* formats, containing the uncompressed PCM data from the digitized source. In recent years, compressed audio file formats have received a great deal of attention, most notably the MP3 (MPEG-1 Audio Layer 3), the Vorbis codec, and the Advanced Audio Coding (AAC) codec. Many of these "lossy" audio formats translate the sound into the frequency domain (using the Fourier transform or a related technique called Linear Predictive Coding) to package the sound in a way that allows compression choices to be made based on the human hearing model, by discarding perceptually irrelevant frequencies in the sound. Unlike the PCM formats outlined above, MP3 files are much harder to encode, manipulate, and process in real time, due to the extra step required to decompress and compress the audio into and out of the time domain.

Synthesis

Digital audio systems typically perform a variety of tasks by running processes in *signal processing networks*. Each node in the network typically performs a simple task that either generates or processes an audio signal. Most software for generating and manipulating sound on the computer follows this paradigm, originally outlined by Max Mathews as the *unit generator* model of computer music, where a map or function graph of a signal processing chain is executed for every sample (or vector of samples) passing through the system. A simple algorithm for synthesizing sound with a computer could be implemented using this paradigm with only three unit generators, described as follows.

First, let's assume we have a unit generator that generates a repeating sound waveform and has a controllable parameter for the frequency at which it repeats. We refer to this piece of code as an oscillator. Most typical digital oscillators work by playing back small tables or arrays of PCM audio data that outlines a specific waveform. These *wavetables* can contain incredibly simple patterns (e.g., a sine or square wave) or complex patterns from the outside world (e.g., a professionally recorded segment of a piano playing a single note).

If we play our oscillator directly (i.e., set its frequency to an audible value and route it directly to the D/A) we will hear a constant tone as the wavetable repeats over and over again. In order to attain a more nuanced and articulate sound, we may want to vary the volume of the oscillator over time so that it remains silent until we want a sound to occur. The oscillator will then increase in volume so that we can hear it. When we want the sound to silence again, we fade the oscillator down. Rather than rewriting the oscillator itself to accommodate instructions for volume control, we could design a second unit generator that takes a list of time and amplitude instructions and uses those to generate a so-called *envelope*, or ramp that changes over time. Our *envelope generator* generates an audio signal in the range of 0 to 1, though the sound from it is never experienced directly. Our third unit generator simply multiplies, sample per sample, the output of our oscillator with the output of our envelope generator. This *amplifier* code allows us to use our envelope ramp to dynamically change the volume of the oscillator, allowing the sound to fade in and out as we like.

In a commercial synthesizer, further algorithms could be inserted into the signal network—for example, a filter that could shape the frequency content of the oscillator before it gets to the amplifier. Many synthesis algorithms depend on more than one oscillator, either in parallel (e.g., additive synthesis, in which you create a rich sound by adding many simple waveforms) or through *modulation* (e.g., frequency modulation, where one oscillator modulates the pitch of another).

Sampling

Rather than using a small waveform in computer memory as an oscillator, we could use a longer piece of recorded audio stored as an AIFF or WAV file on our computer's hard disk. This *sample* could then be played back at varying rates, affecting its pitch. For example, playing back a sound at twice the speed at which it was recorded will result in

its rising in pitch by an octave. Similarly, playing a sound at half speed will cause it to drop in pitch by an octave.

Most samplers (i.e., musical instruments based on playing back audio recordings as sound sources) work by assuming that a recording has a *base frequency* that, though often linked to the real pitch of an instrument in the recording, is ultimately arbitrary and simply signifies the frequency at which the sampler will play back the recording at normal speed. For example, if we record a cellist playing a sound at 220 hertz (the musical note A below middle C in the Western scale), we would want that recording to play back normally when we ask our sampler to play us a sound at 220 hertz. If we ask our sampler for a sound at a different frequency, our sampler will divide the requested frequency by the base frequency and use that ratio to determine the playback speed of the sampler. For example, if we want to hear a 440 hertz sound from our cello sample, we play it back at double speed. If we want to hear a sound at middle C (261.62558 hertz), we play back our sample at 1.189207136 times the original speed.

Many samplers use recordings that have meta-data associated with them to help give the sampler algorithm information that it needs to play back the sound correctly. The base frequency is often one of these pieces of information, as are *loop points* within the recording that the sampler can safely use to make the sound repeat for longer than the length of the original recording. For example, an orchestral string sample loaded into a commercial sampler may last for only a few seconds, but a record producer or keyboard player may need the sound to last much longer; in this case, the recording is designed so that in the middle of the recording there is a region that can be safely repeated, ad infinitum if need be, to create a sense of a much longer recording.

Effects processing

In addition to serving as a generator of sound, computers are used increasingly as machines for *processing* audio. The field of digital audio processing (DAP) is one of the most extensive areas for research in both the academic computer music communities and the commercial music industry. Faster computing speeds and the increased standardization of digital audio processing systems has allowed most techniques for sound processing to happen in real time, either using software algorithms or audio DSP coprocessors such as the Digidesign TDM and T|C Electronics Powercore cards.

As we saw with audio representation, audio effects processing is typically done using either time- or frequency-domain algorithms that process a stream of audio vectors. An echo effect, for example, can be easily implemented by creating a buffer of sample memory to delay a sound and play it back later, mixing it in with the original. Extremely short delays (of one or two samples) can be used to implement digital filters, which attenuate or boost different frequency ranges in the sound. Slightly longer delays create resonation points called *comb filters* that form an important building block in simulating the short echoes in room reverberation. A variable-delay comb filter creates the resonant swooshing effect called *flanging*. Longer delays are used to create a variety of echo, reverberation, and looping systems and can also be used to create *pitch shifters* (by varying the playback speed of a slightly delayed sound).

Audio analysis

A final important area of research, especially in interactive sound environments, is the derivation of information from audio analysis. Speech recognition is perhaps the most obvious application of this, and a variety of paradigms for recognizing speech exist today, largely divided between "trained" systems (which accept a wide vocabulary from a single user) and "untrained" systems (which attempt to understand a small set of words spoken by anyone). Many of the tools implemented in speech recognition systems can be abstracted to derive a wealth of information from virtually any sound source.

Interactive systems that "listen" to an audio input typically use a few simple techniques to abstract a complex sound source into a control source that can be mapped as a parameter in interaction design. For example, a plot of average amplitude of an audio signal over time can be used to modulate a variable continuously through a technique called *envelope following*. Similarly, a threshold of amplitude can be set to trigger an event when the sound reaches a certain level; this technique of *attack detection* ("attack" is a common term for the onset of a sound) can be used, for example, to create a visual action synchronized with percussive sounds coming into the computer.

The technique of pitch tracking, which uses a variety of analysis techniques to attempt to discern the fundamental frequency of an input sound that is reasonably harmonic, is often used in interactive computer music to track a musician in real time, comparing her/his notes against a "score" in the computer's memory. This technology of score-following can be used to sequence interactive events in a computer program without having to rely on absolute timing information, allowing musicians to deviate from a strict tempo, improvise, or otherwise inject a more fluid musicianship into a performance.

A wide variety of timbral analysis tools also exist to transform an audio signal into data that can be mapped to computer-mediated interactive events. Simple algorithms such as *zero-crossing counters*, which tabulate the number of times a time-domain audio signal crosses from positive to negative polarity, can be used to derive the amount of noise in an audio signal. Fourier analysis can also be used to find, for example, the five loudest frequency components in a sound, allowing the sound to be examined for harmonicity or timbral brightness. Filter banks and envelope followers can be combined to split a sound into overlapping frequency ranges that can then be used to drive another process. This technique is used in a common piece of effects hardware called the *vocoder*, in which a harmonic signal (such as a synthesizer) has different frequency ranges boosted or attenuated by a noisy signal (usually speech). The effect is that of one sound "talking" through another sound; it is among a family of techniques called cross-synthesis.

Music as information

Digital representations of music, as opposed to sound, vary widely in scope and character. By far the most common system for representing real-time musical performance data is the Musical Instrument Digital Interface (MIDI) specification,

released in 1983 by a consortium of synthesizer manufacturers to encourage interoperability between different brands of digital music equipment. Based on a unidirectional, low-speed serial specification, MIDI represents different categories of musical *events* (notes, continuous changes, tempo and synchronization information) as abstract numerical values, nearly always with a 7-bit (0–127) numeric resolution.

Over the years, the increasing complexity of synthesizers and computer music systems began to draw attention to the drawbacks of the simple MIDI specification. In particular, the lack of support for the fast transmission of digital audio and high-precision, syntactic synthesizer control specifications along the same cable led to a number of alternative systems. Open Sound Control, developed by a research team at the University of California, Berkeley, makes the interesting assumption that the recording studio (or computer music studio) of the future will use standard network interfaces (Ethernet or wireless TCP/IP communication) as the medium for communication. OSC allows a client-server model of communication between controllers (keyboards, touch screens) and digital audio devices (synthesizers, effects processors, or general-purpose computers), all through UDP packets transmitted on the network.

The following code examples are written in Processing using Krister Olsson's Ess library (*www.processing.org/reference/libraries*) to facilitate sound synthesis and playback. The Ess library includes classes for audio playback in timed units (`AudioChannel`), playback as a continuous process (`AudioStream`), use of real-time input from the computer's audio hardware (`AudioInput`), and writing of audio output to disk (`AudioFile`). In addition, two classes of unit generator-style functions are available: `AudioGenerators`, which synthesize sound (e.g., `SineWave`, a class that generates a sine waveform), and `AudioFilters`, which process previously generated audio (e.g., `Reverb`, a class to apply reverberation). An `FFT` class (for audio analysis) is also provided.

Example 1, 2: Synthesizer (pp. 593, 594)
These two examples show two different methodologies for synthesizing sound. The first example fills an `AudioStream` with the output of a bank of sine waves (represented as an array of `SineWave` generators). The `audioStreamWrite()` function behaves in a manner analogous to the `draw()` function in the main Processing language, in that it repeats indefinitely to generate the audio by updating the state of the different `SineWave` oscillators and writing them (through the `generate()` method) into the `AudioStream`. The frequency properties of the different `SineWave` generators are set based on the mouse position, which also determines where a snapshot of the audio waveform being generated is drawn to the canvas. The second example shows the use of an `AudioChannel` class to generate a sequence of algorithmically generated synthesized events, which are created by a `TriangleWave` generator filtered through an `Envelope` that fades in and out each "note." The notes themselves are generated entirely in the `setup()` function (i.e., the program is noninteractive), based on a sequence of frequencies provided in a `rawSequence[]` array.

Example 3: Sample playback (p. 595)
The playback of an audio sample can be achieved by instantiating an instance of the `AudioChannel` class with a filename of a sample to read in. This example uses an array of six `AudioChannel` objects, each with the same short sample of a cello (*cela3.aif*). By varying the effective `SamplingRate` of each channel, we can change the playback speed (and as a result, the pitch) of the cello sample when it is sounded (by the `play()` method to the `AudioChannel`). The example shows a simple Pong-like simulation where a sound is triggered at each end of the ball's trajectory as well as when it crosses the center of the canvas. Because the `AudioChannel` playback routine will last for different durations depending on the `SamplingRate` we randomly assign to it, we have no way of guaranteeing that a given `AudioChannel` will be finished playing when the next sound is called for. As a result, a `while()` loop in the code searches through the array of `AudioChannel` objects whenever a sound is called for, querying their state property to see if they are available to play a sound. This demonstrates a simple form of *voice allocation*, an important technique in managing polyphony in systems where we have a finite number of "voices" or sound-generating engines to draw from to make multiple sounds at the same time. An `Envelope` filter is also used to fade the `AudioChannel` in or out as it plays to prevent clicks in the sound.

Example 4: Effects processor (p. 597)
Ess (like many other computer music toolkits) allows for the processing of audio to occur *in-place*; that is, it lets us take a sound, change it in some way, and store it in the same block of memory so that we can continue to process it without having to create a duplicate copy of the audio. This allows us to use effects that rely on some degree of feedback. In this example, we take a sample of electric guitar chords (played by an `AudioChannel` class) and process it through a `Reverb` filter. We then take this (already reverberated) sound and process it again, gradually degenerating the original sound by adding more and more reverberation. A similar technique in the analog domain provides the basis for a well-known piece of the electroacoustic repertoire, Alvin Lucier's 1969 masterpiece "I Am Sitting in a Room."

Example 5: Audio analysis (p. 598)
In addition to classes that provide for the generation and manipulation of audio streams and events, Ess provides an `FFT` class to analyze an `AudioChannel` using the Fast Fourier Transform, filling an array with the spectrum of a particular sound. This allows us to look at the frequency content of the sound we're providing, which we can then use to make decisions in our program or, as in this example, visualize during the `draw()` function as a graph. The code draws two versions of a spectrogram for an `AudioChannel` containing a sound file of a sine sweep: the first (drawn in black) shows the spectrum for the current `FFT` frame (i.e., the sound as we're hearing it now); the second (in white) shows the maximum amplitude achieved in each frequency band so far, gradually decaying over time. This second graph is an example of a *peak hold*, a feature that exists on many audio analysis tools (level meters, etc.) to give an analyst a sense of how the current signal compares to what has come before. In the `draw()`

routine, we plot the FFT channels along a logarithmic space, so that the channels representing lower frequencies are farther apart than the ones representing high frequencies on the right of the canvas; this appropriately approximates our perception of frequency as pitch.

Tools for sound programming

A wide variety of tools are available to the digital artist working with sound. Sound recording, editing, mixing, and playback are typically accomplished through digital sound editors and so-called digital audio workstation (DAW) environments. Sound editors range from open source and free software (MixViews, Audacity) to professional-level two-track mastering programs (BIAS Software's Peak application, Digidesign's Sound Designer). These programs typically allow you to import and record sounds, edit them with clipboard functionality (copy, paste, etc.), and perform a variety of simple digital sound processing (DSP) tasks nondestructively on the sound file itself, such as signal normalization, fading edits, and sample-rate conversion. Often these programs will act as hosts for software plug-ins originally designed for working inside of DAW software.

Digital audio workstation suites offer a full range of multitrack recording, playback, processing, and mixing tools, allowing for the production of large-scale, highly layered projects. DAW software is now considered standard in the music recording and production industry, gradually replacing reel-to-reel tape as the medium for producing commercial recordings. The Avid/Digidesign Pro Tools software, considered the industry standard, allows for the recording and mixing of many tracks of audio in real time along a timeline roughly similar to that in a video NLE (nonlinear editing) environment . Automation curves can be drawn to specify different parameters (volume, pan) of these tracks, which contain clips of audio ("regions" or "soundbites") that can be assembled and edited nondestructively. The Pro Tools system uses hardware-accelerated DSP cards to facilitate mixing as well as to host plug-ins that allow for the high-quality processing of audio tracks in real time. Other DAW software applications, such as Apple's Logic Audio, Mark of the Unicorn's Digital Performer, Steinberg's Nuendo, and Cakewalk's Sonar, perform many of the same tasks using software-only platforms. All of these platforms also support third-party audio plug-ins written in a variety of formats, such as Apple's AudioUnits (AU), Steinberg's Virtual Studio Technology (VST), or Microsoft's DirectX format. Most DAW programs also include extensive support for MIDI, allowing the package to control and sequence external synthesizers, samplers, and drum machines; as well as software plug-in "instruments" that run inside the DAW itself as sound generators.

Classic computer music "languages," most of which are derived from Max Mathews' MUSIC program, are still in wide use today. Some of these, such as CSound (developed by Barry Vercoe at MIT) have wide followings and are taught in computer music studios as standard tools for electroacoustic composition. The majority of these MUSIC-N programs use text files for input, though they are increasingly available with graphical editors for

many tasks. Typically, two text files are used; the first contains a description of the sound to be generated using a specification language that defines one or more "instruments" made by combining simple unit generators. A second file contains the "score," a list of instructions specifying which instrument in the first file plays what event, when, for how long, and with what variable parameters. Most of these programs go beyond simple task-based synthesis and audio processing to facilitate algorithmic composition, often by building on top of a standard programming language; F. Richard Moore's CLM package, for example, is built on top of Common LISP. Some of these languages have been retrofitted in recent years to work in real time (as opposed to rendering a sound file to disk); Real-Time Cmix, for example, contains a C-style parser as well as support for connectivity from clients over network sockets and MIDI.

A number of computer music environments were begun with the premise of real-time interaction as a foundational principle of the system. The Max development environment for real-time media, first developed at IRCAM in the 1980s and currently developed by Cycling'74, is a visual programming system based on a control graph of "objects" that execute and pass messages to one another in real time. The MSP extensions to Max allow for the design of customizable synthesis and signal-processing systems, all of which run in real time. A variety of sibling languages to Max exist, including Pure Data (developed by the original author of Max, Miller Puckette) and jMax (a Java-based version of Max still maintained at IRCAM). James McCartney's SuperCollider program and Ge Wang and Perry Cook's ChucK software are both textual languages designed to execute real-time interactive sound algorithms.

Finally, standard computer languages have a variety of APIs to choose from when working with sound. Phil Burke's JSyn (Java Synthesis) provides a unit generator-based API for doing real-time sound synthesis and processing in Java. The CCRMA Synthesis ToolKit (STK) is a C++ library of routines aimed at low-level synthesizer design and centered on physical modeling synthesis technology.

Ess, a sound library for Processing that has many features in common with the above-mentioned languages, is used in the examples for this text. Because of the overhead of doing real-time signal processing in the Java language, it will typically be more efficient to work in one of the other environments listed above if your needs require substantial real-time audio performance.

Conclusion

A wide variety of tools and techniques are available for working computationally with sound, due to the close integration of digital technology and sound creation over the last half-century. Whether your goal is to implement a complex reactive synthesis environment or simply to mix some audio recordings, software exists to help you fill your needs. Furthermore, sound-friendly visual development environments (such as Max) allow you to create custom software from scratch. A basic understanding of the principles behind digital audio recording, manipulation, and synthesis can be indispensable in order to better translate your creative ideas into the sonic medium. As

the tools improve and the discourse of multimedia becomes more interdisciplinary, sound will become even better integrated into digital arts education and practice.

Notes

1. Douglas Kahn, *Noise, Water, Meat: A History of Sound in the Arts* (MIT Press, 2001), p. 10.
2. Paul Théberge, *Any Sound You Can Imagine: Making Music / Consuming Technology* (Wesleyan University Press, 1997), p. 105.
3. John Cage, "Credo: The Future of Music (1937)," in *John Cage: An Anthology*, edited by Richard Kostelanetz (Praeger, 1970), p. 52.
4. Joel Chadabe, *Electric Sound: The Past and Promise of Electronic Music* (Prentice Hall, 1996), p. 145.
5. Curtis Roads, *The Computer Music Tutorial* (MIT Press, 1996), p. 43.
6. Albert Bregman, *Auditory Scene Analysis* (MIT Press, 1994), p. 213.

Code

Example 1: Synthesizer

```
/**
  Sound is generated in real time by summing together harmonically related
  sine tones. Overall pitch and harmonic detuning is controlled by the mouse.
  Based on the Spooky Stream Save Ess example
*/

import krister.Ess.*;

int numSines = 5;                   // Number of oscillators to use
AudioStream myStream;               // Audio stream to write into
SineWave[] myWave;                  // Array of sines
FadeOut myFadeOut;                  // Amplitude ramp function
FadeIn myFadeIn;                    // Amplitude ramp function

void setup() {
  size(256, 200);
  Ess.start(this);                  // Start Ess
  myStream = new AudioStream();     // Create a new AudioStream
  myStream.smoothPan = true;
  myWave = new SineWave[numSines];  // Initialize the oscillators
  for (int i = 0; i < myWave.length; i++) {
    float sinVolume = (1.0 / myWave.length) / (i + 1);
    myWave[i] = new SineWave(0, sinVolume);
  }
  myFadeOut = new FadeOut();        // Create amplitude ramp
  myFadeIn = new FadeIn();          // Create amplitude ramp
  myStream.start();                 // Start audio
}

void draw() {
  noStroke();
```

```
  fill(0, 20);
  rect(0, 0, width, height);  // Draw the background
  float offset = millis() - myStream.bufferStartTime;
  int interp = int((offset / myStream.duration) * myStream.size);
  stroke(255);
  for (int i = 0; i < width; i++) {
    float y1 = mouseY;
    float y2 = y1;
    if (i+interp+1 < myStream.buffer2.length) {
      y1 -= myStream.buffer2[i+interp] * height/2;
      y2 -= myStream.buffer2[i+interp+1] * height/2;
    }
    line(i, y1, i+1, y2);  // Draw the waves
  }
}

void audioStreamWrite(AudioStream s) {
  // Figure out frequencies and detune amounts from the mouse
  // using exponential scaling to approximate pitch perception
  float yoffset = (height-mouseY) / float(height);
  float frequency = pow(1000, yoffset)+150;
  float detune = float(mouseX)/width-0.5;
  myWave[0].generate(myStream);              // Generate first sine, replace Stream
  myWave[0].phase += myStream.size;          // Increment the phase
  myWave[0].phase %= myStream.sampleRate;
  for (int i = 1; i < myWave.length; i++) {  // Add remaining sines into the Stream
    myWave[i].generate(myStream, Ess.ADD);
    myWave[i].phase = myWave[0].phase;
  }
  myFadeOut.filter(myStream);                // Fade down the audio
  for (int i = 0; i < myWave.length; i++) {  // Set the frequencies
    myWave[i].frequency = round(frequency * (i+1 + i*detune));
    myWave[i].phase = 0;
  }
  myFadeIn.filter(myStream);                 // Fade up the audio
}
```

Example 2: Synthesizer

```
/**
  Sound is generated at setup with a triangle waveform and a simple envelope
  generator. Insert your own array of notes as 'rawSequence' and let it roll.
*/

import krister.Ess.*;

AudioChannel myChannel;   // Create channel
TriangleWave myWave;      // Create triangle waveform
Envelope myEnvelope;      // Create envelope
int numNotes = 200;       // Number of notes
int noteDuration = 300;   // Duration of each note in milliseconds
float[] rawSequence = {
  293.6648, 293.6648, 329.62756, 329.62756, 391.9955, 369.99445, 293.6648, 293.6648,
```

```
  329.62756, 293.6648, 439.99997, 391.9955, 293.6648, 293.6648, 587.3294, 493.8834,
  391.9955, 369.99445, 329.62756, 523.25116, 523.25116, 493.8834, 391.9955,
  439.99997, 391.9955 };  // Happy birthday

void setup() {
  size(100, 100);
  Ess.start(this);                      // Start Ess
  myChannel = new AudioChannel();       // Create a new AudioChannel
  myChannel.initChannel(myChannel.frames(rawSequence.length * noteDuration));
  int current = 0;
  myWave = new TriangleWave(480, 0.3);  // Create triangle wave
  EPoint[] myEnv = new EPoint[3];       // Three-step breakpoint function
  myEnv[0] = new EPoint(0, 0);          // Start at 0
  myEnv[1] = new EPoint(0.25, 1);       // Attack
  myEnv[2] = new EPoint(2, 0);          // Release
  myEnvelope = new Envelope(myEnv);     // Bind Envelope to the breakpoint function
  int time = 0;
  for (int i = 0; i < rawSequence.length; i++) {
    myWave.frequency = rawSequence[current];    // Update waveform frequency
    int begin = myChannel.frames(time);         // Starting position within Channel
    int e = int(noteDuration*0.8);
    int end = myChannel.frames(e);              // Ending position with Channel
    myWave.generate(myChannel, begin, end);     // Render triangle wave
    myEnvelope.filter(myChannel, begin, end);   // Apply envelope
    current++;                                  // Move to next note
    time += noteDuration;                       // Increment the Channel output point
  }
  myChannel.play();                             // Play the sound!
}

void draw() { }  // Empty draw() keeps the program running

public void stop() {
  Ess.stop();    // When program stops, stop Ess too
  super.stop();
}
```

Example 3: Sample playback

```
/**
  Loads a sound file off disk and plays it in multiple voices at multiple sampling
  increments (demonstrating voice allocation), panning it back and forth between
  the speakers. Based on Ping Pong by Krister Olsson <http://tree-axis.com>
*/

import krister.Ess.*;

AudioChannel[] mySound = new AudioChannel[6];  // Six channels of audio playback
Envelope myEnvelope;                           // Create Envelope

boolean left = true;
boolean middle = false;
boolean right = false;
```

```
// Sampling rates to choose from
int[] rates = { 44100, 22050, 29433, 49500, 11025, 37083 };

void setup() {
  size(256,200);
  stroke(255);
  Ess.start(this);  // Start Ess
  // Load sounds and set initial panning
  // Sounds must be located in the sketch's "data" folder
  for (int i = 0; i < 6; i++) {
    mySound[i] = new AudioChannel("cela3.aif");
    mySound[i].smoothPan=true;
    mySound[i].pan(Ess.LEFT);
    mySound[i].panTo(1,4000);
  }
  EPoint[] myEnv = new EPoint[3];     // Three-step breakpoint function
  myEnv[0] = new EPoint(0, 0);        // Start at 0
  myEnv[1] = new EPoint(0.25, 1);     // Attack
  myEnv[2] = new EPoint(2, 0);        // Release
  myEnvelope = new Envelope(myEnv);   // Bind an Envelope to the breakpoint function
}

void draw() {
  int playSound = 0;  // How many sounds do we play on this frame?
  int which = -1;     // If so, on which voice?

  noStroke();
  fill(0, 15);
  rect(0, 0, width, height);          // Fade background
  stroke(102);
  line(width/2, 0, width/2, height);  // Center line
  float interp = lerp(0, width, (mySound[0].pan+1) / 2.0 );
  stroke(255);
  line(interp, 0, interp, height);    // Moving line

  // Trigger 1-3 samples when the line passes the center line or hits an edge
  if ((mySound[0].pan < 0) && (middle == true)) {
    playSound = int(random(1,3));
    middle = false;
  } else if ((mySound[0].pan > 0) && (middle == false)) {
    playSound = int(random(1,3));
    middle = true;
  } else if ((mySound[0].pan < -0.9) && (left == true)) {
    playSound = int(random(1,3));
    left = false;
  } else if ((mySound[0].pan > -0.9) && (left == false)) {
    left = true;
  } else if ((mySound[0].pan > 0.9) && (right == true)) {
    playSound = int(random(1,3));
    right = false;
  } else if ((mySound[0].pan < 0.9) && (right == false)) {
    right = true;
  }
```

```
  // Voice allocation block; figure out which AudioChannels are free
  while (playSound > 0) {
    for (int i = 0; i < mySound.length; i++) {
      if (mySound[i].state == Ess.STOPPED) {
        which = i;  // Find a free voice
      }
    }
    // If a voice is available and selected, play it
    if (which != -1) {
      mySound[which].sampleRate(rates[int(random(0,6))], false);
      mySound[which].play();
      myEnvelope.filter(mySound[which]);  // Apply envelope
    }
    playSound--;
  }

}

public void stop() {
  Ess.stop();  // When program stops, stop Ess too
  super.stop();
}

void audioOutputPan(AudioOutput c) {
  c.panTo(-c.pan, 4000);  // Reverse pan direction
}
```

Example 4: Effects processor

```
/**
  Applies reverb 10 times to a succession of guitar chords.
  Inspired by Alvin Lucier's "I am Sitting in a Room."
  Based on Reverb by Krister Olsson <http://www.tree-axis.com>
*/

import krister.Ess.*;

AudioChannel myChannel;
Reverb myReverb;
Normalize myNormalize;

int numRepeats = 9;
int repeats = 0;
float rectWidth;

void setup() {
  size(256, 200);
  noStroke();
  background(0);
  rectWidth = width / (numRepeats + 1.0);
  Ess.start(this);  // Start Ess
  // Load audio file into a AudioChannel, file must be in the sketch's "data" folder
  myChannel = new AudioChannel("guitar.aif");
```

```
  myReverb = new Reverb();
  myNormalize = new Normalize();
  myNormalize.filter(myChannel);  // Normalize the audio
  myChannel.play(1);
}

void draw() {
  if (repeats < numRepeats) {
    if (myChannel.state == Ess.STOPPED) {  // If the audio isn't playing
      myChannel.adjustChannel(myChannel.size/16, Ess.END);
      myChannel.out(myChannel.size);
      // Apply reverberation "in place" to the audio in the channel
      myReverb.filter(myChannel);
      // Normalize the signal
      myNormalize.filter(myChannel);
      myChannel.play(1);
      repeats++;
    }
  } else {
    exit();  // Quit the program
  }
  // Draw rectangle to show the current repeat (1 of 9)
  rect(rectWidth * repeats, 0, rectWidth-1, height);
}

public void stop() {
  Ess.stop();  // When program stops, stop Ess too
  super.stop();
}
```

Example 5: Audio analysis

```
/**
  Analyzes a sound file using a Fast Fourier Transform, and plots both the current
  spectral frame and a "peak-hold" plot of the maximum over time using logarithmic
  scaling. Based on examples by Krister Olsson <http://tree-axis.com>
*/

import krister.Ess.*;

AudioChannel myChannel;
FFT myFFT;
int bands = 256;  // Number of FFT frequency bands to calculate

void setup() {
  size(1024, 200);

  Ess.start(this);  // Start Ess
  // Load "test.aif" into a new AudioChannel, file must be in the "data" folder
  myChannel = new AudioChannel("test.aif");
  myChannel.play(Ess.FOREVER);
  myFFT = new FFT(bands * 2);  // We want 256 frequency bands, so we pass in 512
}
```

```
void draw() {
  background(176);
  // Get spectrum
  myFFT.getSpectrum(myChannel);
  // Draw FFT data
  stroke(255);
  for (int i = 0; i < bands; i++) {
    float x = width - pow(1024, (255.0-i)/bands);
    float maxY = max(0, myFFT.maxSpectrum[i] * height*2);
    float freY = max(0, myFFT.spectrum[i] * height*2);
    // Draw maximum lines
    stroke(255);
    line(x, height, x, height-maxY);
    // Draw frequency lines
    stroke(0);
    line(x, height, x, height-freY);
  }
}

public void stop() {
  Ess.stop();  // When program stops, stop Ess too
  super.stop();
}
```

Resources

Sound toolkits and resources

Vercoe, Barry, et al. CSound. Synthesis and signal processing language, 1984. http://www.csounds.com.

Garton, Brad, David Topper, et al. Real-Time Cmix. Synthesis and signal processing language, 1995. http://rtcmix.org.

Wang, Ge, and Perry Cook. ChucK. Real-time audio programming language, 2002. http://chuck.cs.princeton.edu.

McCartney, James, et al. SuperCollider. Real-time audio programming language, 1996. http://www.audiosynth.com.

Puckette, Miller, David Zicarelli, et al. Max/MSP. Graphical development environment for music and multimedia, 1986. http://www.cycling74.com.

Puckette, Miller, et al. Pure Data (Pd). Graphical development environment for music and multimedia, 1996. http://www-crca.ucsd.edu/~msp/software.html.

Burke, Phil. JSyn. Java API for real-time audio, 1997. http://www.softsynth.com/jsyn.

Cook, Perry, and Gary Scavone. STK. C++ synthesis toolkit, 1996. http://ccrma.stanford.edu/software/stk.

Lopez-Lezcano, Fernando, maintainer. Planet CCRMA. Collection of open source audio software for Linux, 2005. http://ccrma.stanford.edu/planetccrma/software.

Klingbeil, Michael. SPEAR. Spectral editor, 2004. http://www.klingbeil.com/spear.

Waveform Software. Sox, PVCX, AmberX. Freeware sound conversion / spectral processing / granular synthesis software, 2005. http://www.waveformsoftware.com.

Audacity. Open source audio waveform editor, 2002. http://audacity.sourceforge.net.

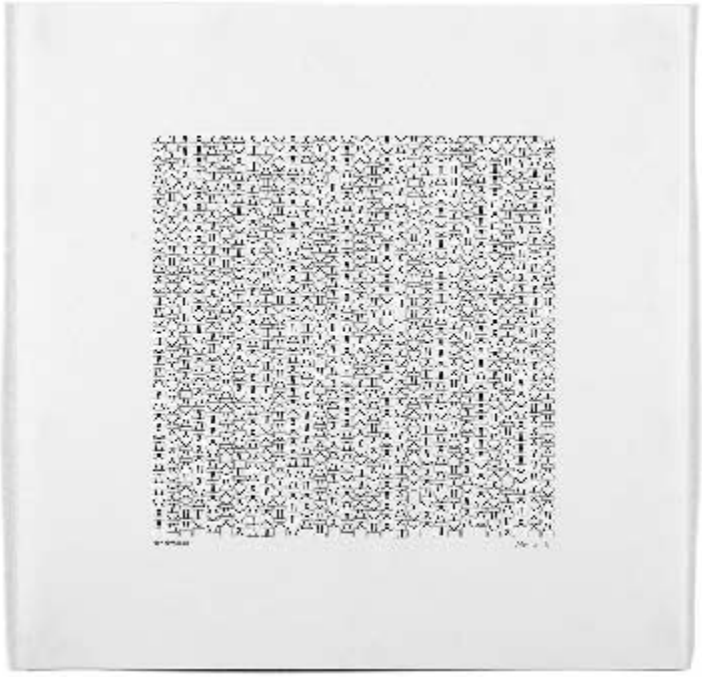
Texts

Bregman, Albert. *Auditory Scene Analysis*. MIT Press, 1994.

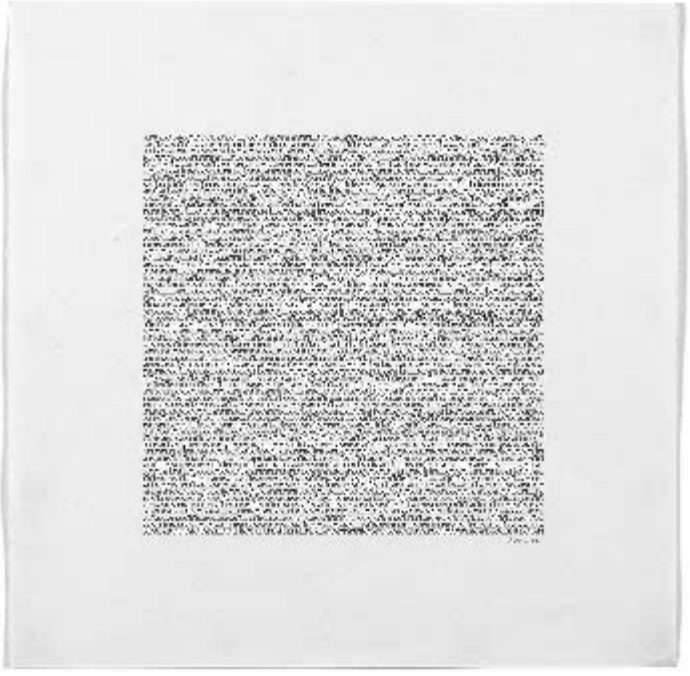
Chadabe, Joel. *Electric Sound: The Past and Promise of Electronic Music*. Prentice Hall, 1996.
Garnett, Guy E. "The Aesthetics of Interactive Computer Music." *Computer Music Journal* 25:1, (2001).
Kahn, Douglas. *Noise, Water, Meat: A History of Sound in the Arts*. MIT Press, 2001.
Lysloff, Rene, and Leslie Gay, eds. *Music and Technoculture*. Wesleyan University Press, 2003.
Maurer, John. "A Brief History of Algorithmic Composition." Stanford University. http://ccrma-www.stanford.edu/~blackrse/algorithm.html.
Paradiso, Joseph A. "American Innovations in Electronic Musical Instruments." *New Music Box* 6. http://www.newmusicbox.org/third-person/oct99.
Prendergast, Mark. *The Ambient Century: From Mahler to Moby—The Evolution of Sound in the Electronic Age*. Bloomsbury, 2003.
Puckette, Miller. *Theory and Techniques of Electronic Music*. University of California, San Diego, 2006. http://crca.ucsd.edu/~msp/techniques.htm.
Rowe, Robert. *Interactive Music Systems*. MIT Press, 1993.
Rowe, Robert. *Machine Musicianship*. MIT Press, 2001.
Theberge, Paul. *Any Sound You Can Imagine: Making Music / Consuming Technology*. Wesleyan University Press, 1997.
Supper, Martin. "A Few Remarks on Algorithmic Composition." *Computer Music Journal* 25:1 (2001).
Winkler, Todd. *Composing Interactive Music: Techniques and Ideas Using Max*. MIT Press, 1998.
Roads, Curtis. *The Computer Music Tutorial*. MIT Press, 1996.
Roads, Curtis. *Microsound*. MIT Press, 2002.

Artists

Aphex Twin (Richard James). http://www.drukqs.net.
Bailey, Chris. http://www.music.columbia.edu/~chris.
Cope, David. http://arts.ucsc.edu/faculty/cope.
DuBois, R. Luke. http://lukedubois.com.
Eno, Brian. http://www.enoweb.co.uk.
Garton, Brad. http://music.columbia.edu/~brad.
Inge, Leif. http://www.notam02.no/9.
Interface (Dan Trueman, Curtis Bahn, Tomie Hahn). http://www.arts.rpi.edu/crb/interface.
Lewis, George. http://kalvos.org/lewisge.html.
Kimura, Mari. http://homepages.nyu.edu/~mk4.
Oliveros, Pauline. http://www.deeplistening.org/pauline.
Oswald, John. http://www.plunderphonics.com.
Risset, Jean-Claude. http://www.cdemusic.org/artists/risset.html.
Schumacher, Michael J. http://www.diapasongallery.org/mjs.page.html.
Sonami, Laetitia. http://www.sonami.net.
Stone, Carl. http://www.sukothai.com.
Truax, Barry. http://www.sfu.ca/~truax.
Vitiello, Stephen. http://www.stephenvitiello.com.
Xenakis, Iannis. http://www.iannis-xenakis.org.



Manfred Mohr. *P-020b*, 1972.
19 ¾" × 19 ¾".
Plotter drawing ink on paper.
Image courtesy
bitforms gallery, nyc.



Manfred Mohr. *P-122d*, 1972.
19 ¾" × 19 ¾".
Plotter drawing ink on paper.
Image courtesy
bitforms gallery, nyc.

Extension 6: Print

Text by Casey Reas

Digital technologies have spawned many changes to printing within the arts. The introduction of laser printers and personal computers into design offices in the mid-1980s was a catalyst for years of experimentation and innovation in typeface design, layout, and printing. Artists have produced prints from software since the mid-1960s, but these techniques have surged since 1990. Innovations have given digitally made prints a longer estimated life than color photographs printed from film. The recent deluge of digital cameras provided another change. Amateurs and professionals are skipping the lab and printing their images at home.

This short text provides a brief history of the digital printing technologies that have led to these new techniques. It presents examples of software written to produce print output, and discusses a few common contemporary print technologies. The industry surrounding digital printing is full of trademarked names and buzzwords, so this text aspires to demystify some of the terminology and provide pointers to additional information. The content that follows is tailored for printing at home or working with a vendor to produce small editions.

Print and computers

When they originated in the 1960s, computer graphics were more often seen printed on paper than on screens. Computers of this time were enormous, expensive machines that were accessible only at research centers and universities, but prints found their way into galleries, journals, and newspapers. In 1963, the *Computers and Automation* journal announced the first competition for computer graphics to be judged using aesthetic criteria.[1] The U.S. Army Ballistic Missile Research Laboratories won the first two competitions, but A. Michael Noll and Frieder Nake won the following two. In 1965 in Stuttgart, Georg Nees and Frieder Nake were the first individuals to exhibit their computer-generated images in a gallery. The same year, the work of A. Michael Noll and Bela Julesz was exhibited at the Howard Wise gallery in New York.[2] These shows presented drawings defined by code and output using a plotter. A plotter is a machine that controls the position of a physical pen on a drawing surface. Nake described his plotter in an essay for the Cybernetic Serendipity exhibition catalog: "I used the Graphomat Zuse Z 64 drawing machine controlled by punch tape. The machine has a drawing head guiding four pens, fed by Indian ink of different colours with nibs of varying thicknesses."[3] Because of mechanical and software limitations, the drawings exhibited in these shows were sparse, mostly geometric, black-and-white images. The plotter remained one of the most common output devices into the 1980s and is still in

use today. Over the years, artists have explored many drawing surfaces and have attached brushes, pencils, and other marking instruments to the plotter's head.

Another area of printed computer graphics produced during the 1960s was more similar to photography than to drawings. At Bell Laboratories, the engineers Kenneth Knowlton and Leon Harmon explored what they called "picture processing." To create their 1966 *Studies in Perception I*, a 5 × 12 foot print of a reclining nude, they scanned a photograph with a device similar to a television camera to convert it to a series of numbers stored on a magnetic tape.[4] The picture's range of gray values was reduced to eight levels, and when it was printed using a microfilm plotter, each gray level was replaced with a corresponding icon with a similar density that, when viewed at a distance, simulated the gray value. The icons used for the print included mathematical symbols (multiplication and division signs) and electronics symbols for diodes and transistors. The final enlargement was made from the microfilm using a photographic process. The techniques used to create this work envisioned the now familiar technologies of scanning and image filtering.

During the 1980s, the cost of computers and printing technology fell to levels within reach of individual artists and designers. The artist Mark Wilson started to work with personal computers and plotters in 1980. He utilized the resolution and precision of the plotter to produce dense, geometric textures. He has continued to explore new printing techniques and has produced work using the plotter to deposit acrylic onto linen and to draw with ink on mylar. In his 1985 book *Drawing with Computers*, Wilson explained how to use a personal computer and the BASIC programming language to control a plotter. The following program from the book draws a line from coordinate (100, 100) to (200, 200). The text following each apostrophe is a comment explaining the purpose of each line:

```
100 OPEN "COM1,1200,0,7,1" AS #1              'Serial communications opened
110 PRINT #1,"!AE"                            'Initialize plotter
120 STARTX-100                                'Create and assign STARTX variable
130 STARTY-100                                'Create and assign STARTY variable
140 ENDX-200                                  'Create and assign ENDX variable
150 ENDY-200                                  'Create and assign ENDY variable
160 PRINT #1,"!AX"+STR$(STARTX)+STR$(STARTY); 'Move pen head to coordinate (100,100)
170 PRINT #1,"!AY"+STR$(ENDX)+STR$(ENDY);     'Draw line to coordinate (200,200)
```

Each plotter manufacturer (e.g., Hewlett-Packard, Tektronix, IBM) had its own commands, but they all provided the ability to move the pen up and down and to move it from one location to another. The above example was written for a Tektronix plotter.

The LaserWriter printer, introduced for Apple's Macintosh computer in 1985, was an important innovation in printing technology. Combined with page-layout software and the Mac's GUI interface, this early laser printer was the catalyst for the desktop publishing revolution. The LaserWriter printed at 300 dots per inch (dpi), while the more common dot matrix printers at that time printed at 72 dpi. The PostScript programming language was the essential software component of the LaserWriter. Each printer had a

processor that ran the PostScript interpreter to rasterize the data for printing. Forms in a PostScript file are defined by coordinates and shape commands. This makes it possible to transform elements within a composition without losing resolution. The PostScript equivalent of the BASIC program presented above is:

```
/startX 100 def             % Create and assign startX variable
/startY 100 def             % Create and assign startY variable
/endX 200 def               % Create and assign endX variable
/endY 200 def               % Create and assign endY variable
startX startY moveto        % Move to coordinate (100,100)
endX endY lineto stroke     % Draw line to coordinate (200,200)
```

Over time, the PostScript language became the de facto standard for printed output, and it served as a basis for ambitious visual experimentation. Rather than programming PostScript files directly, most people used software like Aldus PageMaker to design pages. Graphic designers started to use this technology to assert more personal control over typographic layout and to explore new visual possibilities.

This expanded freedom was manifested in *Emigre* magazine, started in 1984 with Rudy Vanderlans as editor/designer and his partner Zuzana Licko supplying new typefaces. Emigre is a digital type foundry, and the magazine simultaneously promoted the fonts and served as a protagonist in the wider debate on digital aesthetics within the design community. The pages of *Emigre* were filled with portfolios and interviews with the most interesting designers of that time, including P. Scott Makela, Rick Valicenti, and The Designers Republic. In the time before the Web and blogs, *Emigre* magazine was a place to be informed of new ideas and to discuss possible futures. When the last issue was published in 2005, Rick Poynor, founder of *Eye* magazine, wrote: "*Emigre* emerged at a time when technology was changing design forever and the magazine sizzled with this energy and excitement." With regard to Vanderlans, Poynor stated, "His page designs were exemplary demonstrations of the new digital design aesthetic."[5]

In *The End of Print* (1995), the design writer Lewis Blackwell wrote, "The designer of today works with resources that did not exist just a decade (or less) ago. The digital age has transformed the tools available and the processes by which ideas are realized. The functionality of earlier production methods has been emulated and superseded by this new technology."[6] The typographic flexibility encouraged by desktop publishing software was pushed to its limit in the pages of *Beach Culture* and *Ray Gun*. David Carson's designs for these magazines about surfing and music were exciting and controversial. The visual style from page to page was wildly idiosyncratic, and the multiple layers of stressed typography and photography often bordered on illegibility, but the design was in the spirit of the content and the time. Carson explored the extremes of letter spacing, typographic texture, and mixing typefaces, but always with a sensitive eye for composition.

Printed work in the 1980s and 1990s was not all created using commercial desktop publishing software. Many artists and designers wrote custom software to realize their vision. The Beowolf typeface (p. 169) designed by LettError utilized code to produce a font

that randomizes the design of every letter as it is printed. John Maeda's early printed works from the 1990s fostered a surge of interest in programming images for print. These commercial posters for Japanese printers and type foundries utilize algorithms to generate images of astonishing complexity and resolution. Each of the ten posters for Morisawa use only the company's logo to create a diverse range of dense, delicate typographic compositions. The work of Kenneth A. Huff is more representational than the graphic work of Maeda, yet it too is rooted in algorithmic abstraction. Influenced by organic patterns such as those found in lichen and drying mud, Huff develops unique numerical formulas to use as the basis of his compositions. He places an emphasis on accurate rendering of textures and lighting to evoke tactility, but his work diverges from the constraints of physical materials.

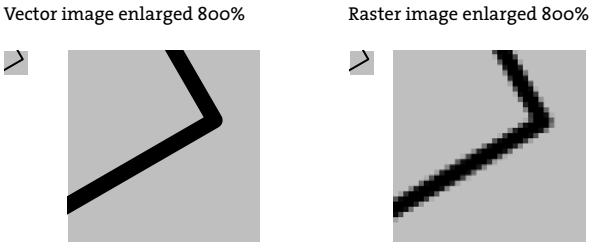
Rapid advancements in inks and paper technology have made it possible for digital prints to have the longevity necessary for the art market. Artists who previously worked with traditional materials and techniques have started to use digital printing technologies in place of etching and lithography, and photographers have replaced darkrooms with computers. In the early 1990s, Iris prints became the first digital prints to be heavily used by established printmakers and photographers. At this time Robert Rauschenberg, well known for his lithographs, began making Iris prints with vegetable dyes and transferring the images to another piece of paper to make collages. This technique was similar to his transferred lithograph images and silkscreen painting from the 1960s, but gave him more control. Other well-known early adopters of digital printing include Chuck Close, Jim Dine, and William Wegman. Artists have also started to combine new digital techniques with traditional art materials. For example, recent prints from Jean-Pierre Hébert use a computer-controlled machine to etch a copper plate, and the physical printing process is executed traditionally. Manfred Mohr prints onto canvas and then stretches the prints over frames.

Subsequent technologies have again changed the landscape of digital printing, and more artists continue to use the technology. The featured software projects by Jared Tarbell (p. 157), Martin Wattenberg (p. 161), and James Paterson (p. 165) are additional examples of excellent work with a focus on printed output.

High-resolution file export

Images saved directly from screen are created at the screen's resolution, typically around 100 pixels per inch (ppi). This low resolution is clearly visible when the images are printed with a high-resolution printer. In contrast to screen resolution, printers are capable of 9600 dpi. Two primary techniques are used to create high-resolution files with software. The first technique saves a *vector* file, and the second saves a *raster* (bitmap) file. The vector technique creates files that store shape information as coordinate points. Common vector formats are PDF, AI, EPS, and SVG. The raster technique stores shape information as pixels. Common raster formats are TIFF, JPEG, TARGA, and PNG. A vector file can be output at any size without loss of resolution, but raster files do not scale gracefully.

The difference is illustrated with this diagram:



Use the vector technique to export line art, type, or shapes that can be printed professionally, published, or printed at very large sizes. It's also helpful to create a file that can be edited further with a program like Inkscape or Adobe Illustrator. Raster images are useful when exporting an image from a program that does not refresh its background each frame. If the image accumulates by adding each new frame to the display window, as in code 26-05 (p. 232) and code 44-02 (p. 415), it may not be possible for vector data to achieve the same effect or it may be too much geometry to store in a single file. A raster file does not represent each visual element separately (it saves it as a series of pixels), so it is editable only by programs like GIMP and Photoshop. A raster file can be printed with as much resolution as a vector file if it is output with a large enough width and height setting to give the file a high resolution when scaled for print. For example, to print a four-inch image at 600 dpi would require `size(2400, 2400)` inside `setup()`. Vector files are eventually rasterized during the printing process, so it's simply a matter of when the rasterizing takes place—whether directly from the program, or inside a professional raster image processor (RIP). The following examples clarify the strengths and weaknesses of each technique.

Example 1: Render to PDF (p. 613)
When `PDF` is used as the third parameter to the `size()` function, the program renders to a PDF file instead of drawing to the display window. The file name is set by a fourth parameter to `size()` and the file is saved to the sketch's folder. Most sketches can be renderered as PDF by simply adding the two parameters to the `size()` command and selecting Sketch -> Import Library -> PDF. Once you do this, you'll no longer see the image on screen as it is running, but it becomes possible to create PDF files at sizes much larger than the screen.

Example 2: Render to screen, export to PDF (p. 613)
This example saves a PDF file while simultaneously drawing to the screen. The `beginRecord()` function opens a new file, and all subsequent drawing functions are echoed to this file as well as to the display window. The `endRecord()` function stops the recording process and closes the file. The `beginRecord()` function requires two parameters; the first is the renderer to use (in this example, `PDF`), and the second is the file name.

Example 3: Save one frame from a continuous program (p. 613)
This example saves a PDF file each time the mouse is pressed. The `boolean` variable `saveOneFrame` is set to `true` when a mouse button is pressed, causing `beginRecord()` to run the next time through `draw()`. At the end of `draw()`, `endRecord()` is run and the variable is set to `false` so another file won't be saved while drawing the next frame. Each PDF file is numbered with the current frame (the number of elapsed frames since the program started).

Example 4: Accumulate many frames into one PDF (p. 614)
This example saves multiple frames drawn to the screen within a single PDF file. The file opens when the *B* key is pressed, and everything drawn in subsequent frames is saved into it, until the *E* key is pressed. The background function is run after `beginRecord()` to clear the background within the PDF document as well as in the display window. This example draws only one new line to the PDF file each frame so the file remains small, but it's possible to write thousands of lines each frame. However, when vector files get very large, computers can have difficulty opening and printing them.

Example 5: Save a TIFF image from a high-resolution off-screen buffer (p. 614)
This example creates a TIFF file larger than the screen and draws into it directly, rather than drawing to the screen. The `createGraphics()` function creates an object from the `PGraphics` class (`PGraphics` is the main graphics and rendering context for Processing). The `beginDraw()` method is necessary to prepare for drawing, then each subsequent drawing function is written into the large raster object. The `endDraw()` and `save()` methods are necessary to complete the file and then save it to the machine so that it can later be viewed in a different program such as GIMP or Photoshop.

Example 6: Scale and segment an image (p. 614)
This example saves a series of image files at screen resolution from a single image enlarged to any dimension. These files can be tiled within an image editor such as Photoshop to create a single, high-resolution file. Another program can also be written to tile the images together automatically. A `scaleValue` of 2 tiles the image to 4 files, a `scaleValue` of 3 tiles the image to 9 files, etc. This example code works only with 2D images.

After a file is generated through software, it is often modified before it is printed. Common changes include tweaking color or changing the weight of the lines after print tests. To make changes, load raster files into a program such as GIMP or Photoshop. Load vector files into a program such as Inkscape, Illustrator, or CorelDRAW.

Production

Like traditional printing technologies, creating a high-quality digital print is a craft that requires knowledge and experience in addition to excellent tools and machines. The

quality of a print is affected by the quality of the printer, ink, and paper, the preparation of the digital file, and the printer settings. Each of these components is introduced below.

Printing technologies

Many different printing technologies are currently in use by artists and designers, and each has unique attributes. This list presents some of the most popular ones. Because printing technology changes rapidly, specific printer models are not discussed.

Laser. Laser printers are exceptional because of their high resolution, speed, and low cost per page. For these reasons they are ubiquitous in office environments. Laser printers use a technology similar to that of photocopiers. When a print is made, a cylindrical drum inside the printer is electrically charged. A high-precision laser is reflected to strike the drum, and it reverses the electrical charge where it hits. When the drum comes in contact with charged toner (small particles of carbon blended with a polymer), the toner is attracted to the drum where the laser hit. A sheet of paper is then passed over the drum and the toner is transferred and then fused to the paper surface with heat.

Inkjet. Inkjet prints are the most common technology for home printers because of their low cost and high image quality. They have also become a dominant technology for professional photographic printing. In comparison to laser printers, inkjet printers are slow, but they can produce images with a much higher resolution (currently up to 9600 dpi). They achieve this resolution by precisely squirting tiny droplets of ink onto the page. For this reason, the actual resolution of an inkjet can't compare directly to that of a laser printer, which is more precise. A 600 dpi inkjet printer may produce distinguishable shapes down to 150 dpi, while a laser printer at 600 dpi maintains precision almost to the full 600 dpi. Each inkjet printer operates using one of three technologies: thermal, piezoelectric, or continuous. The thermal technique forces a drop of ink onto the paper by using heat to cause a tiny steam explosion within a chamber. The piezoelectric technique bends a piezoelectric crystal to force a droplet out of the ink chamber. The continuous technique sends a constant flow of droplets but charges each with a varying electrostatic field that determines how or whether it will hit the paper. Inkjet printers are sometimes also called bubblejet printers.

Digital chromogenic print (C Print). A digital C print is similar to a traditional color photographic print, but a digital file is used instead of an enlarged negative. It is made by exposing photographic paper to light (either an LED or laser) inside the printer. It is then processed using chemicals, just like a photographic print. Chromira and Lightjet are two popular types of printers that use this technique.

Iris. Iris printers were developed to make full-color proofs before a job was printed in large numbers on a commercial press. They started to be used as fine art printers in the early 1990s. The most distinct aspect of this technology is the ability to print on many different flexible substrates including paper, silk, and canvas. To make an Iris print, the substrate is attached to a 35-inch-wide metal drum and spun at a high speed while tiny drops of ink are applied to the substrate in a high-pressure stream. An Iris printer is a specific type of inkjet printer.

Giclée. Giclée (pronounced zhee-CLAY) is not a specific printing technology; it is a

term used to define a high-quality, digitally produced fine art print. The term giclée was selected to distance the technique from connotations to digital technology and computers, as a tactic to gain acceptance within the art community. The first giclée prints were made using the Iris printing technology, but they are now made using other technologies as well.

Other techniques. In addition to the printing technologies mentioned above, commercial printers offer a wide selection of specialized printers for creating large-format prints for buses, billboards, and buildings. These printing techniques produce lower resolution, but they look crisp at a distance.

Ink

The various printing technologies use different types of inks. Laser printers use toner, inkjet printers use liquid inks, and C prints don't use inks because they are made with a photographic process. Toner must be fixed with heat, but liquid inks are absorbed into the paper. Inkjet inks can be divided into two broad categories: dye-based and pigmented. In comparison to pigmented inks, the more common dye-based inks are not water-resistant, are less expensive, are less resistant to fading, and can create more vivid colors.

Ink selection partially determines how long a print will last. Inkjet inks have been notorious for fading color (early prints could fade within six months), but prints from some of the current generation of printers are rated to hold their color for over 100 years when used with special papers. Ultraviolet (UV) light is a huge contributing factor to fading. Frame prints with UV-filtering glass or plexiglass to significantly increase their life. According to Wilhelm Imaging Research, an independent research company that rates specific inks and paper, Epson's highest-quality pigmented inks will last for well over 100 years if framed with archival materials under UV-filtering glass and will last for over 200 years in dark storage. Dye-based inks from Epson, Canon, and HP typically maintain color for 10 to 70 years.

Paper

Paper is defined by its surface, material, and weight. Basic surface options include matte, luster, semigloss, gloss, and supergloss with textures ranging from extremely smooth to rough. Paper is typically made from wood fibers, but fibers from cotton, hemp, linen, and rice are also used. Paper made from wood is naturally acidic, so look for papers that are acid-free (pH neutral) to increase the longevity of prints. Paper made from 100 percent rag is the most stable and will not grow brittle with age. The weight of a paper affects its thickness. Fine art papers are usually measured in units of grams per square meter (gsm or g/m^2). Common weights range from 110 (thin) to 350 (thick).

The selection of papers available for digital printing is extremely limited in comparison to those manufactured for traditional fine art and commercial printing, but the selection is still broad. Printer manufacturers such as Hewlett-Packard, Canon, and Epson offer their own papers, but the finest-quality paper can arguably be found at companies that specialize in making paper. Companies like Hahnemühle and Somerset have recently started producing papers specifically for digital printing technologies.

Some inkjet printers require paper to be coated with a special layer to stop the ink from bleeding into the paper and dulling the color. Iris printers offer the widest selection of printable media. An Iris printer can print on any absorbent material and still produce strong color.

Paper for printers comes in sheets and rolls, but depending on where you live, it will either be sized according to the international ISO 216 standard (A4, A3, etc.) or North American sizes (letter, tabloid, etc.).

File preparation

The format, resolution, and color profile are the most important components of preparing a file. For the best printing results, files should be saved in a format that does not compress the data in a way that loses information. For example, the JPEG image format compresses a file by removing color data. Each type of printer produces the best results when files are at a specific resolution. Images should typically be saved at 300 dpi or higher. Images produced for Inkjet printers should prepared at a dpi resolution that is an increment of the maximum printer resolution. For example, an image file prepared for a 2880 dpi printer should be saved at 360 dpi (360 × 8 = 2880). The 360 dpi resolution is suggested for photographic images, but the quality of an image with fine lines can be improved by doubling the resolution to 720 dpi (720 × 4 = 2880). A higher dpi resolution won't help a great deal (with inkjet) and will significantly increase file size. The color profile for a file is essential to match color when working with other people or across different computers. Each file should be tagged with a color profile to specify the color space of the document (e.g., ColorMatch RGB or Adobe RGB 1998). A color profile tells an output device such as a monitor or printer how to interpret the file's numerical color data to display it correctly on that device.

Our experience has shown that complex vector files should be rasterized in a program such as Adobe Photoshop before they are sent to a vendor for printing. Most printers specialize in printing photographs and have more competence in working with image formats like TIFF and PSD. Printers eventually rasterize the image, and it's a good idea to have complete control over this process, unless the print shop specifies otherwise.

Inkjet printer settings

C prints and Iris prints are typically made through a vendor and laser printing is straightforward. Making a high-quality print with an inkjet printer, however, is often done at home but requires following a few important protocols. Because every type of paper behaves differently with each printer, it's necessary to define the paper type within the print dialog box. Most printers offer selections for their proprietary papers. If you are using one of these papers, select it from the list. If you are using a different paper, you may want to install the ICC profile for the paper that matches the printer.[7] The dialog box will also offer an option to set the dpi of the print. Sometimes it's necessary to go into advanced settings to gain access to this. There are a few things to consider when selecting the dpi. Glossy papers can hold a higher resolution than matte papers, and higher-resolution prints require more ink. Unless your print has extremely precise details and will be viewed close up, 1440 dpi is an adequate resolution. Some

printers also have an option to print "high speed." This option can cause visible banding in areas of flat color and should not be used for final prints.

Making test prints is an essential step toward producing a high-quality print. Precise colors and line weights look very different on screen than on paper. If working with a vendor, always have a small test print made to check the quality before producing the final print.

Conclusion

Only within the last fifteen years have digital printing technologies begun to rival traditional printing techniques in their resolution and longevity, but there is a vast discrepancy between the physical qualities of a digital print and an original Hokusai (1760–1849) woodblock print or an Albrecht Dürer (1472–1528) etching. Each printing technique has its constraints and advantages. One arguable advantage of digital printing is the absence of a physical representation of the print. A lithograph has a corresponding stone, and offset printing has a metal plate, but the information for a digital print is stored in a computer's memory and is therefore easier to modify. The primary advantage of digital printing is the ease with which a software image can be manifested as a physical image. In contrast to an image on screen, a print can have a much higher resolution and will exist long after the software becomes incompatible with future computers. There are also disadvantages to printing software images. The ability to animate the image is lost, and the color palette is reduced. Digital printing technology evolves rapidly, and it's possible that the current research into electronic paper (e-paper, e-ink) and organic light-emitting diodes (OLEDs) will allow the best aspects of printed images to merge with the best elements of digital screens. These or other emerging technologies may make it possible for digital printing to evolve into an original area of image making rather than simply mimicking traditional printing technologies.

Notes

1. H. W. Franke, *Computer Graphics, Computer Art* (Phaidon, 1971), p. 60.
2. Ibid., p. 69.
3. Frieder Nake, "Notes on the Programming of Computer Graphics," In *Cybernetic Serendipity*, edited by Jasia Reichardt (Praeger, 1969), p. 77.
4. This work is also credited as Mural in the exhibition catalog *Cybernetic Serendipity*.
5. Rick Poynor, "*Emigre*: An Ending," *Design Observer*, 10 November 2005. http://www.designobserver.com/archives/007816.html.
6. Lewis Blackwell, *The End of Print: The Graphic Design of David Carson* (Chronicle Books, 1995), p. 173.
7. ICC profiles are files that define the mappings between a print's data and the specific paper and printer specifications. Well-designed profiles can increase the quality of a print. Check the website of the paper manufacturer to see if it has created one for your printer.

Code

Example 1: Render to PDF

```
import processing.pdf.*;          // Import PDF code

size(600, 600, PDF, "line.pdf");  // Set PDF as the renderer
background(255);
stroke(0);
line(200, 0, width/2, height);    // Draw line to PDF
exit();                           // Stop the program
```

Example 2: Render to screen, export to PDF

```
import processing.pdf.*;          // Import PDF code

size(600, 600);
beginRecord(PDF, "line.pdf");     // Start writing to PDF
background(255);
stroke(0, 20);
strokeWeight(20);
line(200, 0, 400, height);        // Draw line to screen and to PDF
endRecord();                      // Stop writing to PDF
```

Example 3: Save one frame from a continuous program

```
import processing.pdf.*;  // Import PDF code

boolean saveOneFrame = false;

void setup() {
  size(600, 600);
}

void draw() {
  if (saveOneFrame == true) {             // When the saveOneFrame boolean is true,
    beginRecord(PDF, "line-####.pdf");    // start recording to the PDF
  }
  background(255);
  stroke(0, 20);
  strokeWeight(20);
  line(mouseX, 0, width-mouseY, height);
  if (saveOneFrame == true) {             // If the PDF has been recording,
    endRecord();                          // stop recording,
    saveOneFrame = false;                 // and set the boolean value to false
  }
}

void mousePressed() {                     // When a mouse button is pressed,
  saveOneFrame = true;                    // trigger PDF recording within the draw()
}
```

Example 4: Accumulate many frames into one PDF

```
import processing.pdf.*;  // Import PDF code

void setup() {
  size(600, 600);
  background(255);
}

void draw() {
  stroke(0, 20);
  strokeWeight(20);
  line(mouseX, 0, width-mouseY, height);
}

void keyPressed() {
  if (key == 'B' || key == 'b') {         // When 'B' or 'b' is pressed,
    beginRecord(PDF, "lines.pdf");        // start recording to the PDF
    background(255);                      // Set a white background
  } else if (key == 'E' || key == 'e') {  // When 'E' or 'e' is pressed,
    endRecord();                          // stop recording the PDF and
    exit();                               // quit the program
  }
}
```

Example 5: Save a TIFF image from a high-resolution off-screen buffer

```
PGraphics big;  // Declare a PGraphics variable

void setup() {
  big = createGraphics(3000, 3000, JAVA2D);  // Create a new PGraphics object
  big.beginDraw();                           // Start drawing to the PGraphics object
  big.background(128);                       // Set the background
  big.line(20, 1800, 1800, 900);             // Draw a line
  big.endDraw();                             // Stop drawing to the PGraphics object
  big.save("big.tif");
}
```

Example 6: Scale and segment an image

```
// Draws an image larger than the screen by tiling it into small sections.
// The scaleValue variable sets amount of scaling: 1 is 100%, 2 is 200%, etc.

int scaleValue = 3;  // Multiplication factor
int xoffset = 0;     // x-axis offset
int yoffset = 0;     // y-axis offset

void setup() {
  size(600, 600);
  stroke(0, 100);
}
```

```
void draw() {
  scale(scaleValue);
  translate(xoffset * (-width/scaleValue), yoffset * (-height/scaleValue));
  line(10, 150, 500, 50);
  line(0, 600, 600, 0);
  setOffset();
}

void setOffset() {
  save("lines-" + xoffset + "-" + yoffset + ".jpg");
  xoffset++;
  if (xoffset == scaleValue) {
    xoffset = 0;
    yoffset++;
    if (yoffset == scaleValue) {
      exit();
    }
  }
  background(204);
}
```

Resources

Blackwell, Lewis. *The End of Print: The Graphic Design of David Carson*. Chronicle Books, 1995.

The Designers Republic. Artist website. http://thedesignersrepublic.com.

Mohr, Manfred. Artist website. http://www.emohr.com.

Hébert, Jean-Pierre. Artist website. http://hebert.kitp.ucsb.edu.

Huff, Kenneth A. "Visually Encoding Numbers Utilizing Prime Factors." In *Aesthetic Computing*, edited Paul A. Fishwick. MIT Press, 2006.

Greiman, April. *Hybrid Imagery: The Fusion of Technology and Graphic Design*. Watson-Guptill, 1990.

Nake, Frieder. "Notes on the Programming of Computer Graphics." In *Cybernetic Serendipity*, edited by Jasia Reichardt. Praeger, 1969.

Knowlton, Kenneth A. "Portrait of the Artist as a Young Scientist." In *The Anthology of Computer Art: Sonic Acts XI*, edited by Arie Altena and Lucas van der Velden. Sonic Acts Press, 2006.

Nees, Georg. *Generative Computergraphik*. Siemens AG, 1969.

Noll, Michael A. "The Digital Computer as a Creative Medium." In *Cybernetics, Art and Ideas*, edited by Jasia Reichardt. New York Graphic Society, 1971.

Rick Poynor. "*Emigre*: An Ending." *Design Observer*, 10 November 2005. http://www.designobserver.com/archives/007816.html.

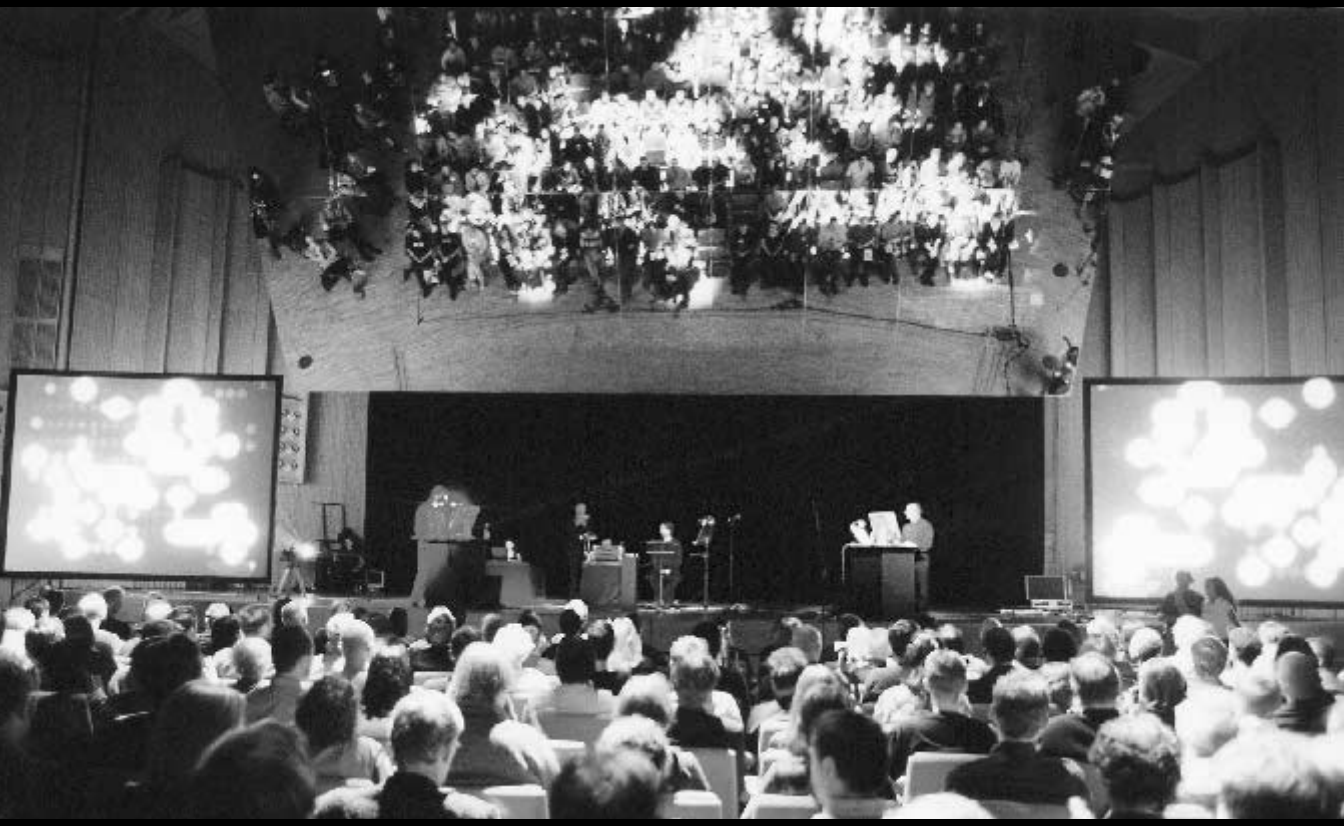
Reichardt, Jasia, ed. *Cybernetic Serendipity: The Computer and the Arts*. Praeger, 1969.

Reid, Glenn C. *Thinking in PostScript*. Addison-Wesley, 1990. Available online at http://www.rightbrain.com/pages/books.html.

Valicenti, Rick. Thirst/3st. Website. http://www.3st.com.

Vanderlans, Rudy, Zuzana Licko, et al. *Emigre: Graphic Design into the Digital Realm*. Van Nostrand Reinhold, 1993.

Wilson, Mark. *Drawing with Computers*. Putnam, 1985.



Extension 7: Mobile

Text by Francis Li

The same relentless technological advancement that has resulted in shrinking the room-filling mainframes of yesteryear to the desktop personal computers of today is still at work, bringing the computational power of those same desktop computers to the handheld computers of tomorrow. Already the "smart" mobile telephone of today has the processing power of the first generation of desktop computers. However, the mobile phone is poised to reach a far greater audience. It has rapidly become a point of access to the Internet and will play a similar role in future interactive services. For good and for bad, the mobile phone is changing the way people communicate and interact with one another, both in the electronic and the physical worlds. It is not uncommon now to exchange text messages instead of email, to hear phones ringing in a movie theater, or even to see a person seemingly talking to herself while walking down the street, using a wireless headset to carry on a voice conversation with a remote partner.

Mobile software applications

The mobile phone has become the platform and subject for new forms of interactive applications and electronic art. The rapid adoption and popularity of the low-cost Short Messaging Service (SMS) in Europe made the ability to exchange short text messages a standard feature on all mobile phones. Software that can receive and respond to text messages have enabled services like Google SMS, in which search queries can be sent to the popular Internet search engine using a text message with the results delivered in a text message response. Sending a text message to Google's social networking service Dodgeball will broadcast your location to all your friends via text message. Like Friendster and other Web-based social networking services, Dodgeball connects friends, friends of friends, and strangers within its member community. Using the mobile phone as its interface allows Dodgeball to go further by enabling opportunistic face-to-face encounters in the physical world. While these services may find the 160-character maximum in a text message to be a limitation, writers contributing to the *CityPoems* project used it as a creative challenge. Physical signs called PoemPoints posted in the city of Leeds, U.K. encouraged members of the community to both contribute and read poems via SMS by sending text messages with PoemPoint codes to the *CityPoems* phone number.

However, as popular as text messaging may be, it will never completely replace the original function of the mobile phone, which is talking with another person. And, being able to receive a phone call at any place and any time means the ringing of a phone can be heard at any place and any time. From simple sequences of synthesized tones to full stereo digital recordings of music and sound clips, sounds from mobile phones are now a

Dialtones: A Telesymphony, 2001. Performed and composed by Golan Levin, Scott Gibbons, and Gregory Shakar. Images courtesy of Golan Levin.

commonplace, if not always welcome, addition to our everyday soundscape. At the 2001 Ars Electronica Festival, Golan Levin and his collaborators premiered the *Dialtones* ring-tone symphony, which used the mobile phones of audience members in a live performance. By way of carefully choreographed calls to audience members, the symphony transformed the mobile phone and its ring tone from an intrusion into an instrument. *Telephony*, first exhibited by Thomson and Craighead in 2000, brought similar themes to a gallery installation of 42 preprogrammed mobile phones arranged in a grid on a wall. Gallery visitors initiated a musical performance of ring tones by dialing into one of the phones, which in turn dialed other phones in a chain reaction.

Playing sound is one way in which the mobile phone can directly affect its surrounding environment. However, it can also be used as a live remote-control interface to its surrounding environment, such as during a *SimpleTEXT* performance where audience members share control of an audio and video projection. *SimpleTEXT* uses the contents of text messages sent in real time during a performance as keywords for image searches and as an input for a generative MIDI music system. In Germany, the mobile phone was used as the interface to play the electronic game Pong as displayed on the side of a building in the *Blinkenlights* project. By calling the *Blinkenlights* phone number, players outside the building in Berlin could use the number keys to move the paddles in the classic game. The ubiquity of the mobile phone allows almost anyone to become a participant in these projects from the comfort and intimacy of their own personal interface.

The mobile phone is also becoming a remote interface by which the physical environment can be "annotated" by electronic content. Code words, numbers, and computer-readable tags can be used to mark places in the physical world that have retrievable electronic content. The *Yellow Arrow* project uses phone numbers printed on highly visible stickers to send and receive text messages related to the physical location of the sticker. The *[murmur]* project uses phone numbers and codes marked on signs in the city of Toronto for retrieving audio recordings. These types of systems are being used for tour guide applications, for organizing games, and for linking all forms of text and imagery including poetry, stories, and photography.

Most of these applications utilize existing capabilities of mobile phones, such as voice dialing, text messaging, and basic Internet browsing. An Internet server or other computer is used to process most of the input and output, and the mobile phone serves merely as a delivery device. Being able to connect to other devices and sources of information is one of the most important characteristics of a mobile phone, but the ability to write custom software that can be installed on a mobile phone allows for more interaction than sending and receiving messages or calls.

Games were some of the first custom software written for mobile phones. In 1997, the Finnish manufacturer Nokia embedded the game Snake into a mobile phone. Although not the first incarnation of the game, after being embedded into more than 350 million handsets, it is arguably the most successful and popular. With its blocky and pixellated visuals, Snake is most commonly praised for its simple and addictive gameplay and has since launched a revival of classic, now considered "retro," games for mobile phones. However, Snake has continued to evolve and take advantage of the mobile phone as a platform in subsequent versions, adding downloadable levels and a

worldwide community leader board of high scores, and the latest version provides multiplayer interaction and color 3D graphics. Games are one of the largest markets for downloadable content on the mobile phone.

Custom software installed on a mobile phone can communicate with a wider range of hardware devices and technologies. For example, global positioning system (GPS) receivers and other location-sensing technologies are being used to query the location of the mobile phone and its user in the physical world. Mapping and wayfinding systems have been the first such applications to be developed, from companies like TeleNav and Wayfinder Systems; location-based service applications like a local business search loom closely on the horizon.

Bluetooth is a wireless networking standard that is used to communicate with devices like GPS receivers, headsets, and other mobile phones. Moreover, a mobile phone with Bluetooth can automatically detect the presence of those devices when they are in its immediate vicinity. This is being used for new social networking applications like the Nokia Sensor. Sensor detects the presence of other mobile phones running the same software and allows the exchange of personal profiles as a way of facilitating face-to-face meetings with strangers.

The mobile platform

Writing custom software for a mobile phone opens possibilities for new applications. But, as with any platform, there are technical and physical constraints. This section presents a broad overview of the characteristics of mobile phones and the ways in which they differ from desktop computers. At the time of this writing, a typical mobile phone has many characteristics in common with a desktop computer from the 1980s, including memory and storage capacity. However, mobile phones are rapidly evolving, and hundreds of new models are being introduced each year around the world. Many of the technical limitations will likely become irrelevant, but the physical constraints of size and weight will remain. After all, a mobile phone must remain mobile!

Application size and storage

Mobile phones have much less storage capacity for software and data than desktop computers. As a result, some phones have limits on the size of applications. If the size of an application exceeds the limit, the phone will refuse to install it. Depending upon the make and model of the mobile phone, the application size limit can be as little as 64,000 bytes. In contrast, a typical compact disc can store 10,000 times as much. Multimedia files including images and sounds are usually the largest contributor to application size, so they must be used in moderation in mobile phone applications. Depending upon the format of the image files, they can be compressed with the same techniques used to optimize images for the Web.

Similarly, data that is saved by the application can also be subject to a size limitation. The total limit on data can be as little as 16,000 bytes on older phones. This is enough to store 80 SMS text messages at 200 bytes each or 8 Email messages at 2,000

bytes each. However, a single digital photo with dimensions of 640 × 480 pixels stored in the JPEG format, common from cameras in mobile phones, can require 20,000 bytes or more of storage. Newer mobile phones generally remove both of these limitations and have storage capacities measured in millions of bytes.

Memory

Mobile phones can have as little as 256 kilobytes of memory for running applications, the same as an original IBM Personal Computer from 1981. Memory is used to store the program code of an application while it is running as well as any additional data files it needs, including multimedia files. If a mobile phone runs out of memory while running an application, the application will often simply terminate without explanation. Again, multimedia files including images and sounds are often the largest consumers of memory. If different images are to be displayed on the screen at the same time, they must be able to fit in memory together with the running program code.

Key input

The primary input mechanism on a mobile phone is its keys. Most mobile phones follow the standard 4 rows by 3 columns key arrangement to allow the input of the numbers 0 though 9, #, and *. This grid arrangement makes it easy to map the keys to directions like up (2), down (8), left (4), and right (6). However, some mobile phones have alternative key arrangements or completely different input mechanisms such as scroll wheels. In addition, most mobile phones have a completely separate key or joystick, similar to a video game controller, that allows four-way directional input.

Most mobile phones also have two or more physical keys placed next to the bottom of the screen, called softkeys. Softkeys can be used for any function by an application and are usually labeled with descriptive text or an icon on the screen just above each key. Softkeys are commonly used as shortcuts for commands and to switch to different screens in an application user interface.

There is no standard for inputting text on a mobile phone. Most text input techniques involve pressing a key multiple times to cycle through a set of characters associated with the key. However, the association of characters and symbols with keys printed on the face of the mobile phone can differ from one mobile phone to the next. Often, it is also necessary to switch to or overlay a different screen on top of an application in order to input text.

Most key arrangements are designed to be used with only one hand, and usually only the thumb of the hand. As a result, pressing multiple keys simultaneously or in very rapid succession is usually not supported. Applications, usually games, that rely on fast reactions and precise timing can sometimes be difficult to support.

Screen output

Mobile phones can have screens of varying sizes, resolutions, and aspect ratios. Some common pixel dimensions include 96 × 64 pixels, 128 × 128 pixels, and 176 × 208 pixels. Although most mobile phones now have color screens, there can be both black-and-white screens and color screens that display 256, 4096, or 65,536 colors.

Multimedia input and output

Most mobile phones have music synthesizers used for playing ring tones. Applications can commonly play individual tones specified by frequency and duration and more complex musical arrangements stored in the MIDI file format. Newer mobile phones can play digitized audio formats including WAV and MP3 and can sometimes also record audio. However, there is often little sound control beyond starting and stopping playback and adjusting the volume. Also, most mobile phones cannot play more than one sound file at a time, often with delays between loading files and starting playback.
Vibrators are commonly used as an alternative to ring tones. Applications can also use vibration as a form of haptic output to the user, such as when losing points in a game or to call attention to an important message.

Mobile phones with digital cameras can be used by an application to capture still images or video. The images and video can be stored or transferred to another device. Computer vision techniques can potentially be used to process the images as an input, for example, recognizing numbers from barcodes or detecting motion for movement in a game.

Networking and the Internet

Mobile phones connect to the Internet via radio-tower antennas maintained by wireless network operators. Speeds are comparable to dial-up modem connections on current-generation networks and can reach broadband speeds and beyond on newer networks. Connection time delays, speed, and reliability vary wildly based on radio tower reception and usage. Internet data passes through servers managed by the network operators that monitor the amount of data being transferred, commonly billed to the user at a per-kilobyte rate. Network operators can restrict access to the Internet, and connections can easily be lost after a short period of inactivity. As a result, it is not easily possible, or practical from a cost perspective, to maintain a continuous connection to a server or other device on the Internet via the mobile phone network.

Close-range wireless networks, also called personal area networks (PAN), are becoming more common on mobile phones as a way to communicate with other mobile phones, desktop computers, and a growing variety of accessories including wireless headsets and GPS receivers. Infrared and Bluetooth are the most common PAN technologies. Data exchanged on a PAN are independent of the mobile phone network.

Preinstalled applications

Most mobile phones include a suite of applications that are pre-installed and cannot be deleted. These applications usually include a Web browser capable of displaying different forms of XML- or HTML-based content. Personal information, including appointments and contacts, is commonly stored and managed by calendar and address book applications on a mobile phone. On some mobile phones, other applications are also allowed to access this information. Other applications include message editors for sending SMS or Email; games; and utilities like alarm clocks.

Programming for mobile phones

Mobile Processing is a tool for programming applications for mobile phones. It uses the same programming language as Processing with additions and omissions that are specific to mobile phone features. Applications created using Mobile Processing run on mobile phones that support Java technology, also known as Java-powered mobile phones. The core language and API of Mobile Processing will run on any Java-powered mobile phone, but not all such phones support the additional features included as libraries. The following examples highlight some of the significant differences between Processing and Mobile Processing and demonstrate how to use features specific to the mobile platform.

Mobile Processing uses a software simulator to run and test mobile phone applications on the desktop computer during development. When ready, the application can be copied to a mobile phone using a PAN connection such as Bluetooth, or uploaded to a Web server for download over the Internet. For some mobile phones, the only way to install an application is to download it from a link on a Web page using the mobile phone's Web browser.

Example 1: Drawing to the screen (p. 626)
Applications on a mobile phone should fill the entire screen. There is no concept of "windows" on most mobile phones, and because most built-in phone applications will use the entire screen for their interfaces, users will generally have the same expectation of other applications. As a result, Mobile Processing does not have a `size()` function to set the width and height of the output. Instead, a Mobile Processing application will automatically fill the entire screen, and the `width` and `height` variables will be assigned the pixel dimensions of the screen. Instead of using absolute coordinates when drawing to the screen, use the `width` and `height` variables to calculate relative positions that will automatically adjust to the size of the mobile phone screen. Mobile Processing supports loading and displaying images stored in the Portable Network Graphics (PNG) format. PNG files can be optimized by most image editing programs to create the smallest possible file size. The format supports 8-bit alpha transparency values, but most phones only support 1-bit alpha transparency, where pixels are either completely transparent or completely opaque.

Example 2, 3: Key presses (pp. 626, 627)
Handling key presses in Mobile Processing is similar to doing it in Processing, but there are additions for keys specific to the mobile phone. By default, Mobile Processing creates a softkey called Exit to allow the user to end the application. A second softkey can be assigned using the `softkey()` function to specify a text label for the key. When the softkey is pressed, the `softkeyPressed()` callback function is called with the label as a parameter. The `textInput()` function will open a screen that allows input using the built-in text entry methods of the mobile phone. The `draw()` loop, if running, will be paused and the new screen will completely replace the output of the application. The text entry screen will look different depending on the make and model of the mobile

phone. After the input is complete, the text will be returned as a `String` and the `draw()` loop will resume. If you do not wish to switch to a new screen in order to gather text input, you can use custom key-handling functions that implement a text entry method called multitap. Pressing a key multiple times in rapid succession cycles through the characters associated with the key. Since multiple key presses can be necessary to interpret a single character, a buffer is used to accumulate and interpret key presses. Use the `multitap()` function to begin interpreting key presses and accumulating characters in the key buffer. The `multitapText` variable is a `String` object containing the accumulated characters. The `multitapDeleteChar()` function deletes a single character from the buffer, and the `multitapClear()` function deletes all the accumulated characters. Use the `noMultitap()` function to stop the buffering process.

Example 4: Networking (p. 627)
Mobile Processing applications use the `PClient` object to connect to the Internet. It is included as part of the core languge of Mobile Processing, and all mobile phones that have an Internet data included with their service plan should be able to use it. Direct connections to computers on the Internet are not supported. Instead, requests can be made to Web servers using the same Hypertext Transport Protocol (HTTP) used by Web browsers. Create a `PClient` object by specifying the name of the server you wish to communicate with. Then, use the `GET()` function to request data. The `GET()` function is used to retrieve the contents of a file or the output of a server-side script.

The `GET()` function returns a `PRequest` object immediately while the phone attempts to establish a connection in the background. In the meantime, the main loop continues and you can draw or animate the status of the connection, if you wish. If the connection is successful, the `libraryEvent()` function will be called by the request object to notify you. At that time, you can then fetch the data by calling `readBytes()`. Again, the phone will attempt to read the data in the background and the main loop will continue. When all of the data is fetched, the request object will call libraryEvent() again and the data can be processed as an array of bytes.

To send data to the server, you can call the `POST()` function on the `PClient` object. The server receives the data as if it were submitted from a form on a Web page. The server can then process the data and send a reply, which can be read using the same `readBytes()` function.

Example 5, 6, 7: Sound (pp. 629, 630)
Sound is available as a library in Mobile Processing. Not all mobile phones will be able to run applications with the Sound library included. If a mobile phone cannot run the Sound library, the application will likely not start at all, or it will terminate without explanation. All mobile phones that can run applications with the Sound library will be able to play synthesized tones. To play a tone, use the `playTone()` function in the `Sound` object, specifying the MIDI note and duration of the tone. To play other types of sound, use the `loadSound()` function to get a `Sound` object. Most mobile phones can play MIDI music files, and some can play WAV or MP3 digitized audio files. Use the

`supportedTypes()` function in `Sound` to return an array of `String` objects listing the supported sound types using Internet MIME codes.

Example 8: Controlling the phone (p. 630)
The Phone library provides control for mobile phone-specific features. These include vibrating the phone, flashing the backlight of the screen, launching the Web browser, and dialing a phone number. Not all phones will support this library or implement all the features. To use it, create a new `Phone` object and then call the function for the feature you wish to control. The `vibrate()` and `flash()` functions take one parameter describing the length of the vibration or backlight flashing, respectively, in milliseconds. The `call()` function takes a `String` containing the phone number to dial, and the `launch()` function takes a `String` containing the URL to fetch.

Mobile Processing includes additional libraries that support even more mobile phone features, including Bluetooth, XML data parsing, and video playback and frame capture. These libraries and even more contributions from the community of Mobile Processing users can be found on the Web.

Mobile programming platforms

The development options for mobile phones are as diverse as the possible applications. Mobile Processing is just one option for writing software applications for mobile phones.

Messaging
Projects like *CityPoems* and services like Dodgeball use text messaging to communicate with a server that does all the computational work. The server software can be developed with almost any language, but implementation of the messaging can require a service contract with a messaging gateway and fees for messages sent and received. Wireless network operators can provide unique phone numbers, called shortcodes, that have fewer digits than typical phone numbers to make the service number more memorable for users. Since all mobile phones support text messaging, these services are accessible to all users without requiring them to install any applications.

Browsers
Most mobile phones include some form of Web browser. Any Web server can easily be configured to serve the mobile-specific XML- or HTML-based content formats currently being used, including the Compact HTML format used by NTT DoCoMo's i-Mode service and the xHTML Mobile and Wireless Markup Language (WML) documents specified by the Wireless Application Protocol (WAP) Forum. These formats support a reasonable subset of HTML features, including images, tables, and forms, but generally lack client-side scripting features such as JavaScript. Server applications can be developed using the same scripting or programming languages used to create Web applications. Although there is no mechanism for plug-ins in these mobile Web browsers, Adobe Flash Lite is

being included in many new phones as a player for the popular vector-based content format. Flash Lite is also one of many players that can display content authored using the Scalable Vector Graphics (SVG) specification, a standard XML-based markup format for vector graphics content.

Runtime environments
A runtime environment allows a software application to run unmodified in the same way on different operating systems and hardware. Mobile Processing is built on top of the Java 2 Micro Edition (J2ME) platform from Sun Microsystems, an application runtime environment based on the same Java programming language and runtime environment used on desktop computers. Development for J2ME can be performed using any Java development tools, including the Eclipse and NetBeans integrated development environments (IDEs), many of which include specific support for J2ME development. Sun provides the Sun Java Wireless Toolkit (WTK), which Mobile Processing uses for building and running applications. Manufacturers like Nokia and Sony Ericsson often provide custom WTK implementations that better approximate the look and feel and functionality of their mobile phones. New libraries for Mobile Processing can be written using these tools. Alternative runtime environments include the Binary Runtime Environment for Wireless (BREW) from Qualcomm and Mophun from Synergenix. The trade-off for providing an environment that can run across diverse devices is performance and a lowest-common-denominator feature set. The latest mobile phone features are usually not immediately available in a runtime environment.

Operating systems
Most operating systems for mobile phones are closed and proprietary, which means that it is not possible to write custom software applications for them (instead, a runtime environment is usually provided). However, operating systems that support custom software development are available for mobile phones and include Symbian, Windows Mobile, Palm OS, and Linux. Developing applications for these operating systems allows for the most flexibility and highest performance at the expense of limiting the application to running on a smaller number of mobile phones. Symbian is owned by a consortium of mobile phone manufacturers including Nokia, Sony Ericsson, and Panasonic, and its operating system can be found running many advanced mobile phones from a wide variety of manufacturers. Windows Mobile and Palm OS have a long history of development as the operating systems for the Pocket PC and Palm personal digital assistants, respectively, and are relative newcomers to the mobile phone market. Although largely unseen in the market at the time of this writing, the Linux operating system is an attractive option because of its open source licensing model.

Conclusion

Mobile phones are an emerging platform for new services and applications that have the potential to change the way we live and communicate. Mobile Processing provides a

way to write custom software applications for mobile phones that can utilize the features built into the hardware as well as connect to the Internet and communicate with servers that provide additional computational power and data. As the ability to rapidly prototype and explore new possibilities for interaction and visualization becomes greater, so do the opportunities for finding innovative new services and applications. The Mobile Processing project aims to help drive this innovation and keep pace with the rapid developments in technology while also increasing the audience of potential designers and developers through the tool itself and the open sharing of ideas and information.

Code

These examples are written for the Mobile Processing programming environment and require a mobile phone that can run Java programs. To download the software, visit the Mobile Processing website: *http://mobile.processing.org*. Like Processing, it is free and open source. Further instructions are included with the software and are available on the website.

Example 1: Drawing to the screen

```
// The image file, named sprite.png in this example, must be located in the
// sketch data folder. From the Sketch menu, choose "Add File" to copy files into
// the sketch data folder.
PImage img = loadImage("sprite.png");
// The coordinates (0, 0) refer to the top-left corder of the screen
image(img, 0, 0);
// The following coordinate calculations will center the image in the screen
image(img, (width - img.width) / 2, (height - img.height) / 2);
// Finally, the next line will position the image in the bottom-right corner
image(img, width - img.width, height - img.height);
```

Example 2: Key presses, using `textInput()`

```
String s;
PFont font;

void setup() {
  font = loadFont();       // Load and set the default font for drawing text
  textFont(font);
  softkey("Input");        // Create a softkey called Input
  s = "No input";          // Initialize s with an initial message
}

void draw() {
  background(200);
  text(s, 0, height / 2);  // Draw the String s in the middle of the screen
}
```

```
void softkeyPressed(String label) {
  // Check the value of the softkey label to determine the action to take
  if (label.equals("Input")) {
    // If the Input softkey is pressed, open a textInput window for the user
    // to type text. It will be drawn on the screen by the draw() method
    s = textInput();
  }
}
```

Example 3: Key presses, using `multitap()`

```
PFont font;
void setup() {
  font = loadFont();
  textFont(font);
  softkey("Delete");  // Use softkey to to delete characters from the multitap buffer
  multitap();         // Turn on multitap key input
}

void draw() {
  background(200);
  text(multitapText, 0, height / 2);  // Draw the text captured with multitap
}

void softkeyPressed(String label) {
  if (label.equals("Delete")) {
    multitapDeleteChar();  // Delete a character
  }
}
```

Example 4: Networking

```
// The PClient object is used to initiate requests to the server
PClient c;
// The PRequest object represents an active request from which we receive
// status information and data from the server
PRequest request;

int counter;
PFont font;
PImage img;
String version;
String error;

void setup() {
  font = loadFont();  // Load and set the default font for drawing text
  textFont(font);
  fill(0);
  // Create a new network connection to connect to the Mobile Processing website
  c = new PClient(this, "mobile.processing.org");
  // Start by fetching the logo for Mobile Processing the filename is a relative path
```

```
  // specified in the same way as a URL in a webpage
  request = c.GET("/images/mobile.png");
  // Use the counter to keep track of what we're fetching
  counter = 0;
}

void draw() {
  background(255);
  int y = 0;
  if (error != null) {
    // A network error has occurred, so display the message
    y += font.baseline;
    text(error, 0, y);
  } else if (img == null) {
    // The img is not yet fetched, so draw a status message
    y += font.baseline;
    text("Fetching image...", 0, y);
  } else {
    // Draw the image
    image(img, (width - img.width) / 2, y);
    y += img.height + font.baseline;
    if (version == null) {
      // The version text is not yet fetched, so draw a status message
      text("Checking version...", 0, y);
    }
    else {
      // Draw the version as reported by the website
      text("Latest version: " + version, 0, y);
    }
  }
}

// The libraryEvent() will be called when a library, in this case the Net
// library, has an event to report back to the program
void libraryEvent(Object library, int event, Object data) {
  // Make sure we handle the event from the right library
  if (library == request) {
    if (event == PRequest.EVENT_CONNECTED) {
      // This event occurs when the connection is complete, so we can start
      // reading the data. The readBytes() method will read all the data returned
      // by the server and send another event when completed.
      request.readBytes();
    } else if (event == PRequest.EVENT_DONE) {
      // Reading is complete! Check the counter to see what we're transferring,
      // then process the data. The data object in this case is an array of bytes.
      byte[] bytes = (byte[]) data;
      if (counter == 0) {
        // This is the logo, so create an image from the bytes
        img = new PImage(bytes);
        // Now that we have the logo image, fetch the latest version text for
        // Mobile Processing. We use the client object to initiate a new request
        request = c.GET("/download/latest.txt");
        // Set the counter to 1 to represent the text
```

```
        counter = 1;
      } else if (counter == 1) {
        // This is the version text, so create a string from the bytes
        version = new String(bytes);
      }
    } else if (event == PRequest.EVENT_ERROR) {
      // The data object in this case is an error message
      error = (String) data;
    }
  }
}
```

Example 5: Sound, using `playTone()`

```
import processing.sound.*;

// Notes range from 0 to 127 as in the MIDI specification
int[] notes = { 60, 62, 64, 65, 67, 69, 71, 72, 74 };

void setup() {
  noLoop();  // No drawing in this sketch, so we don't need to run the draw() loop
}

void keyPressed() {
  if ((key >= '1') && (key <= '9')) {
    // Use the key as an index into the array of notes
    Sound.playTone(notes[key - '1'], 500, 80);
  }
}
```

Example 6: Sound, using `loadSound()`

```
import processing.sound.*;
Sound s;

void setup() {
  // The file, soundtrack.mid, must be copied into the data folder of this sketch
  s = new Sound("soundtrack.mid");
  softkey("Play");
  noLoop();
}

void softkeyPressed(String label) {
  if (label.equals("Play")) {
    s.play();
    softkey("Pause");  // Change the label of the softkey to Pause
  } else if (label.equals("Pause")) {
    s.pause();
    softkey("Play");   // Change the label of the softkey back to Play
  }
}
```

Example 7: Sound, using `supportedTypes()`

```
import processing.sound.*;

PFont font = loadFont();
textFont(font);
background(255);
fill(0);
// Get a list of the supported types of media on the phone
String[] types = Sound.supportedTypes();
// Start at the top of the screen
int y = font.baseline;
// Draw each of the supported types on the screen
for (int i = 0, length = types.length; i < length; i++) {
  // Draw the supported type (represented as an
  // Internet MIME type string, such as audio/x-wav)
  text(types[i], 0, y);
  // Go to the next line
  y += font.height;
}
```

Example 8: Controlling the phone

```
import processing.phone.*;
Phone p;

void setup() {
  p = new Phone(this);
  noLoop();  // No drawing in this sketch, so we don't need to run the draw() loop
}

void keyPressed() {
  switch (key) {
  case '1':
    // Vibrate the phone for 200 milliseconds
    p.vibrate(200);
    break;
  case '2':
    // Flash the backlight for 200 milliseconds
    p.flash(200);
    break;
  case '3':
    // Dial 411 on the phone
    p.call("411");
    break;
  case '4':
    // Launch the Web browser
    p.launch("http://mobile.processing.org/");
    break;
  }
}
```

Resources

Mobile toolkits and references

Adobe. Adobe Flash Lite. http://www.adobe.com/products/flashlite.

Bluetooth. http://www.bluetooth.com.

Microsoft. Windows Mobile. http://www.microsoft.com/windowsmobile.

NetBeans. http://www.netbeans.org.

Nokia Sensor. http://www.nokia.com/sensor/.

NTT. DoCoMo i-Mode. http://www.nttdocomo.co.jp/english/service/imode.

Palm OS. http://www.palmsource.com.

Portable Network Graphics (PNG). http://www.w3.org/Graphics/PNG.

Qualcomm. Binary Runtime Environment for Wireless (BREW). http://brew.qualcomm.com.

Scalable Vector Graphics (SVG). http://www.w3.org/Graphics/SVG.

Sun Microsystems. Java Micro Edition (ME). http://java.sun.com/javame.

Sun Microsystems. Java Wireless Toolkit, http://java.sun.com/products/sjwtoolkit.

Synergenix Interactive. Mophun. http://www.mophun.com.

Symbian OS. http://www.symbian.com.

WAP Forum. http://www.wapforum.org.

Artworks and services

Centrifugalforces. *CityPoems*. SMS and website. February 2003–October 2005. http://www.citypoems.co.uk.

Chaos Computer Club. *Blinkenlights*. SMS-controlled architectural installation, 2001. http://www.blinkenlights.de.

Counts Media. *Yellow Arrow*. SMS Service, 2004. http://yellowarrow.net.

Dodgeball. SMS service, 2001. http://www.dodgeball.com.

Family Filter (Jonah Brucker-Cohen, Tim Redfern, Duncan Murphy). *SimpleTEXT*. SMS Performance, 2003. http://www.simpletext.info.

Levin, Golan, et al. *Dialtones: A Telesymphony*. Performance, 2001. http://www.flong.com/telesymphony/

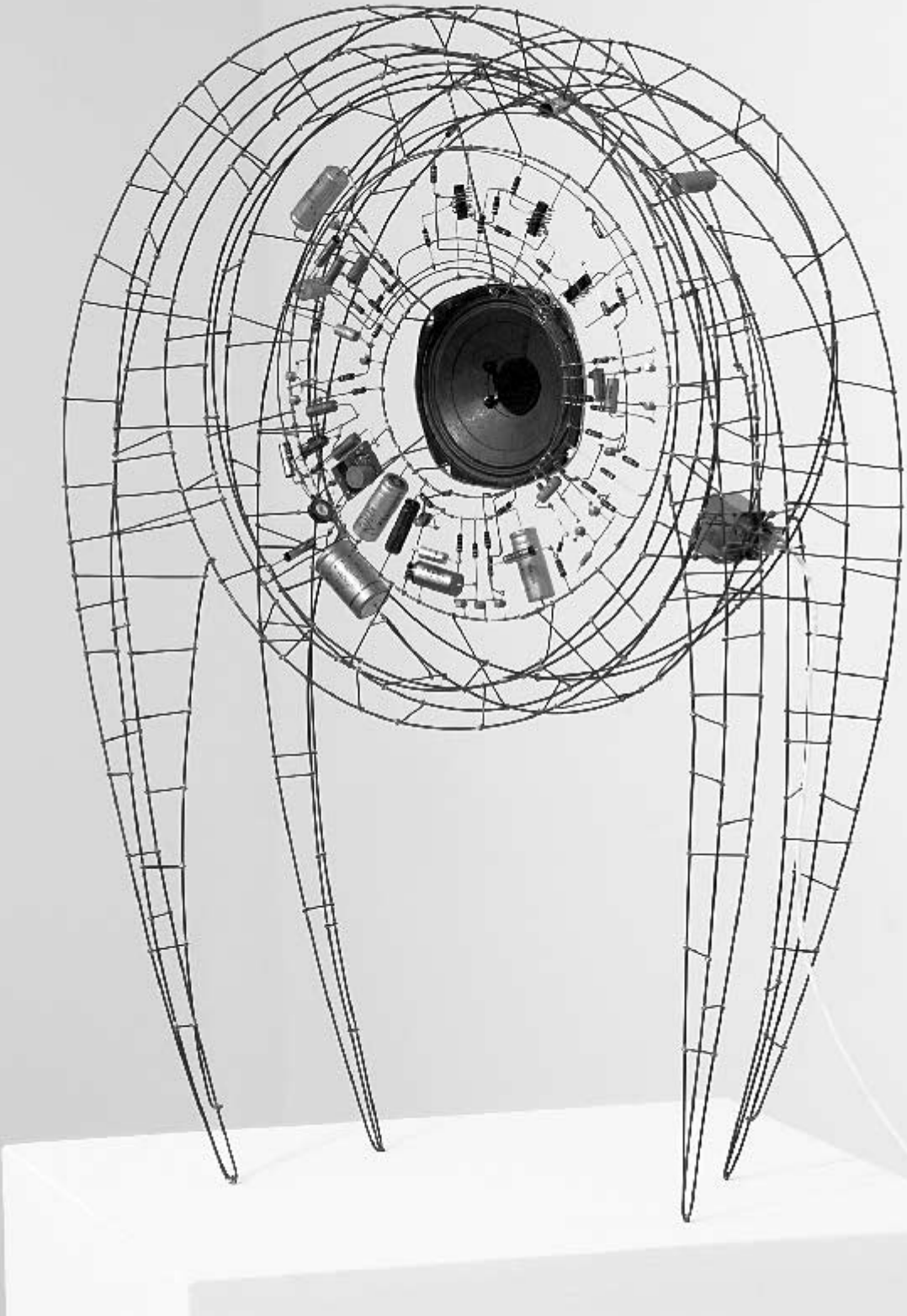
Google SMS. http://www.google.com/sms.

[murmur]. SMS Service, 2004. http://murmurtoronto.ca.

TeleNav. SMS navigation service. http://www.telenav.com.

Thomson, Jon and Alison Craighead. *Telephony*. Mobile phone installation, 2000. http://thomson-craighead.net/docs/telf.html.

Wayfinder Systems. SMS navigation service, 2002. http://www.wayfinder.com.



Extension 8: Electronics

Text by Hernando Barragán and Casey Reas

Software is not limited to running on desktop computers, laptops, and mobile phones. Contemporary cameras, copiers, elevators, toys, washing machines, and artworks found in galleries and museums are controlled with software. Programs written to control these objects use the same concepts discussed earlier in this book (variables, control structures, arrays, etc.), but building the physical parts requires learning about electronics. This text introduces the potential of electronics with examples from art and design and discusses basic terminology and components. Examples written with Wiring and Arduino (two electronics toolkits related to Processing) are presented and explained.

Electronics in the arts

Electronics emerged as a popular material for artists during the 1960s. Artists such as Naum Gabo and Marcel Duchamp had used electrical motors in prior decades, but the wide interest in kinetic sculpture and the foundation of organizations such as Experiments in Art and Technology (E.A.T.) are evidence of a significant emphasis at that time. In The Machine exhibition at The Museum of Modern Art in 1968, Wen-Ying Tsai exhibited *Cybernetic Sculpture*, a structure made of vibrating steel rods illuminated by strobe lights flashing at high frequencies. Variations in the vibration frequency and the light flashes produced changes in the perception of the sculpture. The sculpture responded to sound in the surrounding environment by changing the frequency of the strobe lights. Peter Vogel, another kinetic sculpture pioneer, produced sculptures that generate sound. The sculptures have light sensors (photocells) that detect and respond to a person's shadow when she approaches the sculpture. The sculptures' form is composed directly of the electrical components. The organization of these components forms both the shape of the sculpture and its behavior. Other pioneers during the 1960s include Nam June Paik, Nicolas Schöffer, James Seawright, and Takis.

The range of electronic sculpture created by contemporary artists is impressive. Tim Hawkinson produces sprawling kinetic installations made of cardboard, plastic, tape, and electrical components. His *Überorgan* (2000) uses mechanical principles inspired by a player piano to control the flow of air through balloons the size of whales. The air is pushed through vibrating reeds to create tonal rumbles and squawks. This physical energy contrasts with the psychological tension conveyed through Ken Feingold's sculptures. His *If/Then* (2001) is two identical, bald heads protruding from a cardboard box filled with packing material. These electromechanical talking heads debate their existence and whether they are the same person. Each head listens to the other and forms a response from what it understands. Speech synthesis and recognition software

Peter Vogel. *Circular Structure*, 1979. Speaker, photo cell, circuits, iron wire. 27" × 17 ¾" × 7". Image courtesy of the bitforms gallery, nyc.

are used in tandem with mechanisms to move the face—the result is uncanny. Contemporary projects from Chris Csikszentmihályi (p. 507) and the team of Marc Hansen and Ben Rubin (p. 515) are also featured in this book.

The works of Maywa Denki and Crispin Jones are prototypical of a fascinating area of work between art and product design. Maywa Denki is a Japanese art unit that develops series of products (artworks) that are shown in product demonstrations (live performances). Over the years, they have developed a progeny of creatures, instruments, fashion devices, robots, toys, and tools—all animated by motors and electricity. Devices from the *Edelweiss Series* include *Marmica*, a self-playing marimba that opens like a flower, and *Mustang*, a gasoline-burning aroma machine for people who love exhaust fumes. Crispin Jones creates fully functioning prototypes for objects that are critical reflections of consumer technologies. *Social Mobiles (SoMo)*, developed in collaboration with IDEO, is a set of mobile phones that address the frustration and anger caused by mobile phones in public places. The project humorously explores ways mobile phone calls in public places could be made less disruptive. The *SoMo 1* phone delivers a variable electrical shock to the caller depending on how loud the person at the other end of the conversation is speaking. The ring tone for *SoMo 4* is created by the caller knocking on their phone. As with a knock on a door, the attitude or identity of the caller is revealed through the sound. Other artists working in this area include the Bureau of Inverse Technology, Ryota Kuwakubo, and the team of Tony Dunne and Fiona Raby.

As electronics devices proliferate, it becomes increasingly important for designers to consider new ways to interact with these machines. Working with electronics is an essential component of the emerging interaction design community. The Tangible Media Group (TMG) at the MIT Media Laboratory, led by Hiroshi Ishii, pioneered research into tangible user interfaces to take advantage of human senses and dexterity beyond screen GUIs and clicking a mouse. *Curlybot* (1999) is a toy that can record and play back physical movement. It remembers how it was moved and can replay the motion including pauses and changes in speed and direction. *MusicBottles* (1999) are physical glass bottles that trigger sounds when they are opened. To the person who opens the bottles, the sounds appear to be stored within the bottles, but technically, custom-designed electromagnetic tags allow a special table to know when a bottle has been opened, and the sound is played through nearby speakers. These and other projects from the TMG were instrumental in moving research in interface design away from the screen and into physical space. Research labs at companies like Sony and Philips are other centers for research and innovation into physical interaction design. Academic programs such as New York University's Interactive Telecommunication Program, the Design Interactions course at the Royal College of Art, and the former Interaction Design Institute Ivrea have pioneered educational strategies within in this area.

Electricity

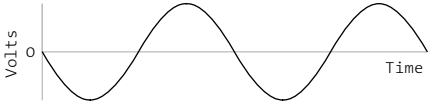
Electricity is something we use daily, but it is difficult to understand. Its effect is experienced in many ways, from observing a light turn on to noticing the battery charge deplete on a laptop computer.

Electrical current is a stream of moving electrons. They flow from one point to another through a conductor. Some materials are better conductors than others. Sticking a fork in a light socket is dangerous because metal is a good conductor and so is your body. The best conductors are copper, silver, and gold. A resistor is the opposite of a conductor. Resistance is the capability of a material to resist the flow of electrons. A substance with a very high resistance is an insulator. Plastic and rubber are very good insulators and for this reason they are used as the protective covering around wires. Voltage is electrical energy—the difference of electrical potential between two points. Current is the amount of electrical energy that flows through a point. Resistance is measured in units called ohms, voltage is measured in volts, and current is measured in amperes (amps). The relation between the three is easiest to understand through an analogy of water flowing through a hose. As explained by the educators Dan O'Sullivan and Tom Igoe:

> The flow of water through a hose is like the flow of electricity through a circuit. Turning the faucet increases the amount of water coming through the hose, or increases the current (amps). The diameter of the hose offers resistance to the current, determining how much water can flow. The speed of the water is equivalent to voltage. When you put your thumb over the end of the hose, you reduce the diameter of the pathway of the water. In other words, the resistance goes up. The current (that is, how much water is flowing) doesn't change, however, so the speed of the water, or voltage, has to go up so that all the water can escape . . . [1]

Electrical current flows in two ways: direct current (DC) and alternating current (AC). A DC signal always flows in the same direction and an AC signal reverses the direction of flow at regular intervals. Batteries and solar cells produce DC signals, and the power that comes from wall sockets is an AC signal:

Direct Current (DC)



Alternating Current (AC)

Depending on your location, the AC power coming into your home is between 100 and 240 volts. Most home appliances use AC current to operate, but some use a transformer to convert the higher-potential AC energy into DC current at smaller voltages. The black plastic boxes (a k a power bricks, power adapters, wall warts) that are used to power laptops or mobile phones are transformers. Most desktop computers have an internal power supply with a transformer to convert the AC signal to the 12-volt and 5-volt DC signals necessary to run the internal electronics. Low voltages are generally safer than high voltages, but it's the amount of current (amps) that makes electricity dangerous.

Components

Electronic components are used to affect the flow of electricity and to convert electrical energy into other forms such as light, heat, and mechanical energy. There are many different components, each with a specific use, but here we introduce only four of the most basic: resistor, capacitor, diode, and transistor.

← *Resistor*
A resistor limits (provides resistance to) the flow of electricity. Resistors are measured in units called ohms. The value 10 ohms is less resistance than 10,000 (10K) ohms. The value of each resistor is marked on the component with a series of colored bands. A variable resistor that changes its resistance when a slider, knob, or dial attached to it is turned is called a potentiometer or trimmer. Variable resistors are designed to change in response to different environmental phenomena. For example, one that changes in response to light is called a photoresistor or photocell, and one that changes in response to heat is called a thermistor. Resistors can be used to limit current, reduce voltage, and perform many other essential tasks.

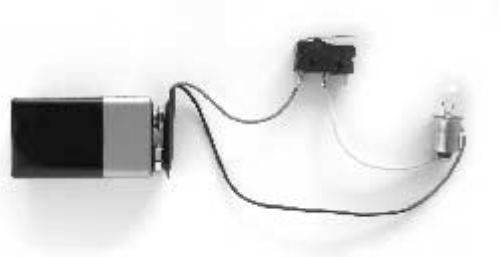
← *Capacitor*
A capacitor stores electrons. It stores electrical charge when current is applied, and it releases charge (discharges) when the current is removed. This can smooth out the dips and spikes in a current signal. Capacitors are combined with resistors to create filters, integrators, differentiators, and oscillators. A simple capacitor is two parallel sheets of conductive materials, separated by an insulator. Capacitors are measured in units called farads. A farad is a large measurement, so most capacitors you will use will be measured in microfarads (μF), picofarads (pF), or nanofarads (nF).

← *Diode*
Current flows only in one direction through a diode. One side is called the cathode (marked on the device with a line) and the other is the anode. Current flows when the anode is more positive than the cathode. Diodes are commonly used to block or invert the negative part of an AC signal. A light-emitting diode (LED) is used to produce light. The longer wire coming out of the LED is the anode and the other is the cathode. LEDs come in many sizes, forms, colors, and brightness levels.

← *Transistor*
A transistor can be used as an electrical switch or an amplifier. A bipolar transistor has three leads (wires) called the base, collector, and emitter. Depending on the type of transistor, applying current to the base either allows current to flow or stops it from flowing through the device from the collector to the emitter. Transistors make it possible for the low current from a microcontroller to control the much higher currents necessary for motors and other power-hungry devices and thus to turn them on and off.

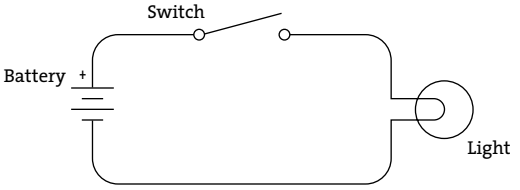
Circuits

An electrical circuit is a configuration of components, typically designed to produce a desired behavior such as decreasing the current, filtering a signal, or turning on an LED. The following simple circuit can be used to turn a light on and off:

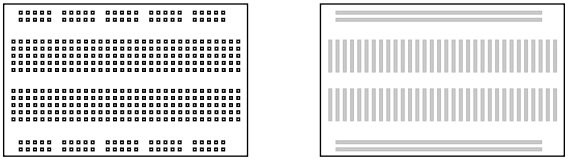


This simple electric circuit is a closed loop with an energy source (battery), a load (lightbulb) that offers a resistance to the flow of electrons and transforms the electric energy into another form of energy (light), wires that carry the electricity, and a switch to connect and disconnect the wires. The electrons move from one end of the battery, through the load, and to the other end.

Circuits are usually represented with diagrams. A circuit diagram uses standardized symbols to represent specific electrical components. It is easier to read the connections on a diagram than on photographs of the components. A diagram of the simple circuit above could look like this:



Circuits are often prototyped on a "breadboard", a rectangular piece of plastic with holes for inserting wires. A breadboard makes it easy to quickly make variations on a circuit without soldering (fusing components together with a soft metal). Conductive strips underneath the surface connect the long horizontal rows at the top and bottom of the board and the short vertical rows within the middle:



Holes in the top surface Internal connections

Circuits are tested with a multimeter, an instrument to measure volts, current, resistance, and other electrical properties. It allows the electrical properties of the circuit

to be read as numbers and is necessary for debugging. Analog multimeters have a small needle that moves from left to right and digital multimeters have a screen that displays numbers. Most multimeters have two metal prongs to probe the circuit and a central dial to select between different modes.

Commonly used circuits are often condensed into small packages. These *integrated circuits* (ICs, or chips) contain dense arrangements of miniaturized components. They are typically small black plastic rectangles with little metal pins sticking out of the sides. Like objects (p. 395) in software, these devices are used as building blocks for creating more complicated projects. ICs are produced to generate signals, amplify signals, control motors, and perform hundreds of other functions. They fit neatly into a breadboard by straddling a gap in the middle.

Microcontrollers and I/O boards

Microcontrollers are small and simple computers. They are the tiny computer brains that automate many aspects of contemporary life through their activities inside devices ranging from alarm clocks to airplanes. A microcontroller has a processor, memory, and input/output interfaces enclosed within a single programmable unit. They range in size from about 1 × 1 cm to 5 × 2 cm. Like desktop computers, they come in many different configurations. Some have the same speed and memory as a personal computer from twenty years ago, but they are much less powerful than current machines, as this comparison tables shows:

Model	Speed	Memory	Cost
Apple Macintosh (1984)	8 MHz	128 Kb	$2500
Atmel ATmega128-8AC Microcontroller (2001)	8 MHz	128 Kb	$15
Apple Mac Mini (2006)	1500 MHz	512,000 Kb	$600

Small metal pins poking out from a microcontroller's edges allow access to the circuits inside. Each pin has its own role. Some are used to supply power, some are for communication, some are inputs, and others can be set to either input or output. The relative voltage at each input pin can be read through software, and the voltage can be set at each output pin. Some pins are reserved for communication. They allow a microcontroller to communicate with computers and other microcontrollers through established communication protocols such as RS-232 serial (p. 645).

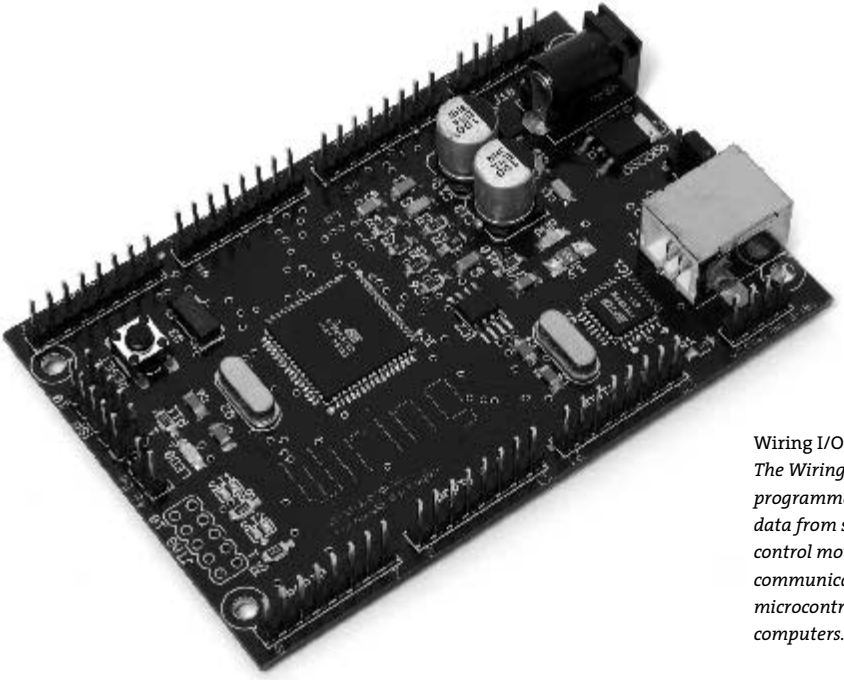
Microcontrollers can be used to build projects directly, but they are often packaged with other components onto a printed circuit board (PCB) to make them easier to use for beginners and for rapid prototyping. We call these boards I/O boards (input/output boards) because they are used to get data in and out of a microcontroller. They are also called micrcontroller modules. We've created three informal groups—bare microcontrollers, programmable I/O boards, and tethered I/O boards—to discuss different ways to utilize microcontrollers in a project.

Bare microcontrollers (PIC, AVR)
Working directly with a bare microcontroller is the most flexible but most difficult way to work. It also has the potential to be the least expensive way of building with electronics, but this economy can be offset by initial development costs and the extra time spent learning how to use it. Microchip PIC and Atmel AVR are two popular families of microcontrollers. Each has variations ranging from simple to elaborate that are appropriate for different types of projects. The memory, speed, and other features affect the cost, the number of pins, and the size of the package. Both families feature chips with between 8 and 100 pins with prices ranging from under $1 to $20. PIC microcontrollers have been on the market for a longer time, and more example code, projects, and books are available for beginners. The AVR chips have a more modern architecture and a wider range of open-source programming tools. Microcontrollers are usually programmed in the C language or their assembly language, but it's also possible to program them in other languages such as BASIC. If you are new to electronics and programming, we don't recommend starting by working directly with PIC or AVR chips. In our experience, beginners have had more success with the options introduced below.

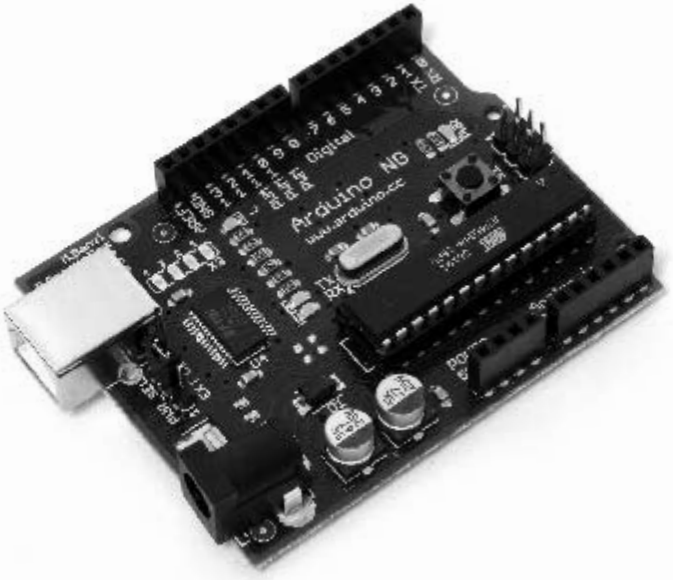
Programmable I/O boards (Wiring, Arduino, Basic Stamp 2, BX-24, OOPic)
A programmable I/O board is a microcontroller situated on a PCB with other components to make it easier to program, attach/detach components, and turn on and off. These boards typically have components to regulate power to protect the microcontroller and a USB or RS-232 serial port to make it easy to attach cables. The small pins on the microcontroller are wired to larger pins called headers that make it easy to insert and remove sensors and motors. Small wires embedded within the PCB connect pins to a corresponding header. Small reset switches make it easy to restart the power without having to physically detach the power supply or battery.

Within the context of this book, the most relevant I/O boards are Wiring ($60) and Arduino ($30). Both were created as tools for designers and artists to build prototypes and to learn about electronics. Both boards use the Wiring language to program their microcontrollers and use a development environment built from the Processing environment. In comparison to the Processing language, the Wiring language provides a similar level of control and ease of use within its domain. They share common language elements when possible, but Wiring has some functions specific to programming microcontrollers and omits the graphics programming functions within Processing. Like Processing programs, Wiring programs are translated into another language before they are run. When a program written with the Wiring language is compiled, it's first translated into the C/C++ language (p. 682) and then compiled using a C/C++ compiler. Wiring is a more powerful system (the board has more memory, I/O pins, serial ports, and other internal capabilities) and is therefore more expensive. The microcontroller on the Wiring board is also directly soldered to the board, while the microprocessor on the Arduino board can be removed and replaced if it is damaged. In the United States, Wiring and Arduino boards are distributed by Spark Fun Electronics.

Both Wiring and Arduino are new boards; others are more established and have a larger user base. The BASIC Stamp 2 ($49) boards from Parallax have a wide user base



Wiring I/O board
The Wiring board can be programmed to read data from sensors, to control motors, and to communicate with other microcontrollers and computers.



Arduino I/O board
The Arduino family of boards works similarly to the Wiring board. There are a range of different Arduino boards, including one that uses Bluetooth and one that is the size of a postage stamp.

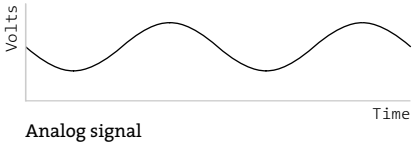
Photographs: John Houck

and are popular microcontrollers for education and for hobbyists, and a vast array of code, project examples, books, and other materials are available. The availability of these materials and user group support makes this system an excellent choice for beginners. The system is sold as a kit including breadboards and other components or as a stand-alone board for more experienced users. BASIC Stamps are programmed using a variation of the BASIC programming language called PBASIC. The NetMedia BasicX-24 ($50) was designed as an updated competitor to the BASIC Stamp 2. The primary advantage of the BasicX-24 is the inclusion of eight ADC pins and a more powerful programming language that allows floating-point math and multitasking. The OOPic ($59–$79) system was designed for robotics and therefore has good support for controlling motors. It includes a development environment for creating user scripts with an object-oriented syntax in different languages like BASIC, C, and Java.

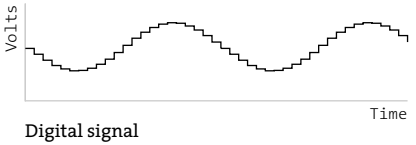
Tethered I/O boards (Teleo, I-Cube X, Phidgets, EZIO)
A tethered I/O board is used to get sensor data into a computer and to control a physical devices (motors, lights, etc.) without the need to program the board. A computer already has many input and output devices such as a monitor, mouse, and keyboard and tethered I/O boards provide a way to communicate between more exotic input devices such as light sensors and video cameras, and output devices such as servomotors and lights. These boards are designed to be easy to use. They often do not require knowledge of electronics because sensors and motors can be directly plugged directly into the board and do not need to interface with other components. Messages are sent and received from the boards through software such as Max/MSP, Flash, Director, and many programming languages. This ease of use often comes at a high price. Four popular systems are the Teleo ($89–$189), the I-Cube X ($299), Phidgets ($10–$380), and EZIO ($129–$179). Teleo is a modular system that can be configured for projects of varying complexity. Each component within the system performs a specific task, and up to 63 can connect together using a special network cable. The basic I-Cube X system includes many basic sensors (touch, proximity) and a wider range are sold separately. It uses the MIDI protocol to communicate with computers or other MIDI devices. Phidgets are an extensive set of USB-compatible boards that simplify bringing sensor data into a computer. They can be interfaced through a range of different development environments and languages. EZIO is a single board with ten digital input/outputs, eight analog inputs, and two PWM (p. 646) outputs.

Sensors and communication

Physical phenomena are measured by electronic devices called sensors. Different sensors have been invented to acquire data related to touch, force, proximity, light, orientation, sound, temperature, and much more. Sensors can be classified into groups according to the type of signals they produce (analog or digital) and the type of phenomena they measure. Analog signals are continuous, but digital signals are discrete and are constrained to a range of values (e.g., 0 to 255):



Analog signal



Digital signal

Most basic analog sensors utilize resistance. Changes in a physical phenomenon modify the resistance of the sensor, therefore varying the voltage output through the sensor. An analog-to-digital converter can continuously measure this changing voltage and convert it to a number that can be used by software. Sensors that produce digital signals send data as binary values to an attached device or computer. These sensors use a voltage (typically between 3.5 and 5 volts) as ON (binary digit 1 or TRUE) and no voltage as a OFF (binary digit 0 or FALSE). More complex sensors include their own microcontrollers to convert the data to digital signals and to use established communication protocols for transmitting these signals to another computer.

Touch and force

Sensing of touch and force is achieved with switches, capacitive sensors, bend sensors, and force-sensitive resistors. A switch is the simplest way to detect touch. A switch is a mechanism that stops or allows the flow of electricity depending on its state, either open (OFF) or closed (ON). Some switches have many possible positions, but most can only be ON or OFF. Touch can also be detected with capacitive sensors. These sensors can be adjusted to detect the touch and proximity (within a few millimeters) of a finger to an object. The sensor can be positioned underneath a nonconductive surface like glass, cardboard, or fabric. This type of sensor is often used for the buttons in an elevator. The QT113H chip from Quantum Technologies packages all of the circuits required for capacitive sensing into an easy-to-interface chip. A bend (flex) sensor is a thin strip of plastic that changes its resistance as it is bent. A force-sensitive resistor (FSR or force sensor) changes its resistance depending on the magnitude of force applied to its surface. FSRs are designed for small amounts of force like the pressure from a finger, and they are available in different shapes including long strips and circular pads.

Presence and distance

There are a wide variety of sensors to measure distance and determine whether a person is present. The simplest way to determine presence is a switch. A switch attached to a door, for example, can be used to determine whether it is open or closed. A change in the state (open or closed) means someone or something is there. Switches come in many different shapes and sizes, but category of small ones called *microswitches* are most useful for this purpose. The infrared (IR) motion detectors used in security systems are another simple way to see if something is moving in the environment. They can't measure distance or the degree of motion, but they have a wide range, and some types can be purchased at hardware stores. IR distance sensors such as the Sharp GP2D family are used to calculate the distance between the sensor and an object. They can calculate distances from 4 to 140 centimeters. The distance is converted into a voltage between 0

and 5 volts that can be read by a microcontroller. Ultrasonic sensors are used for measuring up to 10 meters. This type of device sends a sound pulse and calculates how much time it takes to receive the echo. The Devantech SRF04 is reasonably priced and can determine distances between 8 centimeters and 3 meters.

Light
Sensors for detecting light include photoresistors, phototransistors, and photodiodes. A photoresistor (also called a photocell) is a component that changes its resistance with varying levels of light. It is among the easiest sensors to use. A phototransitor is more sensitive to changes in light and is also easy to use. Photodiodes are also very sensitive and can respond faster to changing light levels, but they are more complex to interface with a microcontroller. Photodiodes are used in the remote control receivers of televisions and stereos.

Position and orientation
A potentiometer is a variable resistor that works by twisting a rotary knob or by moving a slider up and down. The potentiometer's resistance changes with the rotation or up/down movement, and this can affect the voltage level within a circuit. Most rotary potentiometers have a limited range of rotation, but some are able to turn continuously. A tilt sensor is used to crudely measure orientation (up or down). It is a switch with two or more wires and a small metal ball or mercury in a box that touches wires to complete a circuit when it is in a certain orientation. An accelerometer measures the change in movement (acceleration) of an object that it is mounted to. Tiny structures inside the device bend as a result of momentum and the amount of bending is measured. The three-axis ADXL330 accelerometer from Analog Devices works well. Accelerometers are used in cameras to control image stabilization and in automobiles to detect rapid deceleration and release airbags. A digital compass calculates orientation in relation to the earth's magnetic field. The less expensive sensors of this type have a lower accuracy, and they may not work well when situated near objects that emit electromagnetic fields (e.g., motors). The Devantech CMPS03 is a reasonably priced compass sensor.

Sound
A microphone is the simplest and most common device used to detect and measure sound. Sudden changes in volume are the easiest sound elements to read, but processing the sound wave with software (or special hardware) makes it possible to detect specific frequencies or rhythms. A microphone usually requires extra components to amplify the signal before it can be read by a microcontroller. Piezo electric film sensors, commonly used in speakers and microphones, can also be used to detect sound. Sampling a sound wave with a microcontroller can dramatically reduce the quality of the audio signal. For some applications it's better to sample and analyze sound through a desktop computer and to communicate the desired analysis information to an attached microcontroller.

Temperature
A thermistor is a device that changes its resistance with temperature. These sensors are

easy to interface, but they respond slowly to changes. To quantitatively measure temperature, a more sophisticated device is needed. The Dallas DS1820 sensor measures the temperature with high accuracy, but it is more expensive and not as easy to connect to a microcontroller. Flame sensors are more exotic. Devices such as the Hamamatsu UV Tron are tuned to detect open flames such as lighters and candles.

Analog voltage signals from sensors can't be directly interpreted by a computer, so they must be converted it to a digital value. Some microcontrollers provide analog-to-digital converters (ADC or A/D) that measure variations in voltage at an input pin and convert it to a digital value. The range of values depends on the resolution of the ADC; common resolutions are 8 and 10 bits. At 8-bit resolution, an ADC can represent 2^8 (256) different values, where 0 volts corresponds to the value 0 and 5 volts corresponds to 255. A 10-bit ADC provides 1024 different values, where 5 volts corresponds to the value 1023.

Data is sent and received between microcontrollers and computers according to established data protocols such as RS-232 serial, USB, MIDI, TPC/IP, Bluetooth, and other proprietary formats like I2C or SPI. Most electronics prototyping kits and microcontrollers include an RS-232 serial port, and this is therefore a convenient way to communicate. This standard has been around for a long time (it was developed in the late 1960s) and it defines signal levels, timing, physical plugs, and information exchange protocols. The physical RS-232 serial port has largely been replaced in computers by the faster and more flexible (but more complex) universal serial bus (USB), but the protocol is still widely used when combining the USB port with software emulation.

Because a device can have several serial ports, a user must specify which serial port to use for data transmission. On most Windows computers serial port names are `COMx` where `x` can be 1, 2, 3, etc. On UNIX-based systems (Mac OS X and Linux), serial devices are accessed through files in the */dev/* directory. After the serial port is selected, the user must specify the settings for the port. Communication speed will vary with devices, but typical values are 9600, 19,200 and 115,200 bits per second. Once the ports are open for communication on both devices it is possible to send and receive data.

The following examples connect sensors and actuators to a Wiring or Arduino board and communicate the data between the I/O board and a Processing application. When the Wiring and Arduino boards are plugged in to a computer's USB port, it appears on the computer as a serial port, making it possible to send/receive data on it. The Wiring board has two serial ports called *Serial* and *Serial1*; the Arduino board has one called *Serial*. *Serial* is directly available on the USB connector located on the board surface. *Serial1* is available through the Wiring board digital pin numbers 2(Rx) and 3(Tx) for the user's applications.

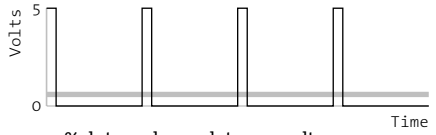
Example 1: Switch (p. 650)
This example sends the status of a switch (ON or OFF) connected to the Wiring or Arduino board to a Processing application running on a computer. Software runs on the board to read the status of a switch connected on digital pin 4. This value 1 is sent to the serial port continuously while the switch is pressed and 0 is sent continuously when the switch is not pressed. The Processing application continuously receives data from the

board and assigns the color of a rectangle on the screen depending on the value of the data. When the switch is pressed the rectangle's color changes from black to light gray.

Example 2: Light sensor (p. 651)
This example brings data from a light sensor (photoresistor) connected to the Wiring or Arduino board's analog input pin 0 into a Processing application running on a computer. Software runs on the board to send the value received from the light sensor to the serial port. Because the light sensor is plugged into an analog input pin, the analog voltage coming into the board is converted into a digital number before it is sent over the serial port. The Processing application changes the color of a rectangle on screen according to the value sent from the board. The rectangle exhibits grays from black to white according to the amount of light received by the sensor. Cover and uncover the sensor with your hand to see a large change.

Controlling physical media

Actuators are devices that act on the physical world. Different types of actuators can create light, motion, heat, and magnetic fields. The digital output pin on a microcontroller can set a voltage of 0 or 5 volts. This value can be used to turn a light or motor on or off, but finer control over brightness and speed requires using a technique called pulse-width modulation (PWM). This is turning a digital output ON and OFF very quickly to simulate values between 0 and 5 volts. If the output is 0 volts for 90% of the time and 5 volts for 10%, this is called a 10% duty cycle. It emulates an analog voltage of 0.5 volts. An 80% duty cycle emulates a 4-volt signal:



10% duty cycle emulates 0.5 volts

80% duty cycle emulates 4 volts

The PWM technique can be used to dim a light, to run a motor at a slow speed, and to control the frequency of a tone through a speaker.

Light
Sending current through a light-emitting diode (LED) is the simplest way to get a microcontroller to control light. An LED is a semiconductor device that emits monochromatic light when a current is applied to it. The color (ranging from ultraviolet to infrared) depends on the semiconductor material used in its construction. LEDs have a wide range of applications from simple blinking indicators and displays to street lamps. They have a long life and are very efficient. Some types of LEDs and high-power LEDs require special power arrangements and interfacing circuits before they can be used with microcontrollers. Incandescent, fluorescent, and electroluminescent light sources always require special interfacing circuits before they can be controlled.

Motion

Motors are used to create rotational and linear movement. The rated voltage, the current drawn by the motor, internal resistance, speed, and torque (force) are factors that determine the power and efficiency of the motor. Direct current (DC) motors turn continuously at very high speeds and can switch between a clockwise and counterclockwise direction. They are usually interfaced with a gear box to reduce the speed and increase the power. Servomotors are modified DC motors that can be set to any position within a 180-degree range. These motors have an internal feedback system to ensure they remain at their position. Stepper motors move in discrete steps in a both directions. The size of the steps depends on the resolution of the motor. Solenoids move linearly (forward or back instead of in circles). A solenoid is a coil of wire with a shaft in the center. When current is applied to the coil, it creates a magnetic field that pulls or pushes the shaft, depending on the type. Muscle wire (shape memory alloy or nitinol), is a nickel-titanium alloy that contracts when power is applied. It is difficult to work with and is slower than motors, but requires less current and is smaller. DC and stepper motors need special interfacing circuits because they require more current than a microcontroller can supply through its output pins. H-bridge chips simplify this interface.

Switches

Relays and transistors are used to turn on and off electric current. A relay is an electromechanical switch. It has a coil of wire that generates a magnetic field when an electrical current is passed through. The magnetic field pulls the two metal contacts of the relay's switch together. Solid-state relays without moving parts are faster than electromechanical relays. Using relays makes it possible to turn ON and OFF devices that can't be connected directly to a microcontroller. These devices include home appliances, 120-volt light bulbs, and all other devices that require more power than the microcontroller can provide. Transistors can also behave like switches. Because they operate electronically and not mechanically, they are much faster than relays.

Sound

Running a signal from a digital out or PWM pin to a small speaker is the easiest way to produce a crude, buzzing noise. For more sophisticated sounds, attach these pins to tone-generator circuits created with a 555 timer IC, capacitors, and resistors. Sound can be recorded and played back using chips such as the Winbond ISD4002. The SpeakJet chip is a sound synthesizer that can synthesize speech by configuring its stored phonemes.

Temperature

Temperature can be controlled by a Peltier junction, a device that works as a heat pump. It transforms electricity into heat and cold at the same time by extracting thermal energy from one side (cooling) into the other side (heating). It can also work in reverse, applying heat or cold to the proper surface to produce an electrical current. Because this device consumes more current than a microcontroller can handle in an output pin, it must be interfaced using transistors, relays, or digital switches like the ones described above.

The following examples demonstrate how to control lights and motors attached to an I/O board through a Processing program:

Example 3: Turning a light on and off (p. 653)
This example sends data from a Processing program running on a computer to a Wiring or Arduino board to turn a light ON or OFF. The program continually writes an *H* to the serial port if the cursor is inside the rectangle and writes a *L* if it's not. Software running on the board receives the data and checks for the value. If the value is *H*, it turns on a light connected to the digital I/O pin number 4, and if the value is *L*, it turns off the light. The light always reflects the status of the rectangle on the computer's screen.

Example 4: Controlling a servomotor (p. 654)
This example controls the position of a servomotor through an interface within a Processing program. When the mouse is dragged through the interface, it writes the position data to the serial port. Software running on a Wiring or Arduino board receives data from the serial port and sets the position of a servomotor connected to the digital I/O pin number 4.

Example 5: Turning a DC motor on and off (p. 655)
This example controls a DC motor from a Processing program. The program displays an interface that responds to a mouse click. When the mouse is clicked within the interface, the program writes data to the serial port. Software running on the board receives data from the serial port and turns the DC motor connected to the PWM pin ON and OFF. The DC motor is connected to the board through an L293D chip to protect the microcontroller from current spikes caused when the motor turns on.

Conclusion

Electronic components and microcontrollers are becoming more common in designed objects and interactive artworks. Although the programming and electronics skills required for many projects call for an advanced understanding of circuits, a number of widely used and highly effective techniques can be implemented and quickly prototyped by novices. The goal of this text is to introduce electronics and to provide enough information to encourage future exploration. As you pursue electronics further, we recommend that you read *CODE* by Charles Petzold to gain a basic understanding of how electronics and computers work, and we recommend that you read *Physical Computing* by Dan O'Sullivan and Tom Igoe for a pragmatic introduction to working with electronics. *Practical Electronics for Inventors* by Paul Scherz is an indispensable resource, and the *Engineer's Mini Notebook* series by Forrest M. Mims III is an excellent source for circuit designs. The Web is a deep resource for learning about electronics and there are many excellent pages listed below in Resources. The best way to learn is by making projects. Build many simple projects and work through the examples in *Physical Computing* to gain familiarity with the different components.

Notes

1. Dan O'Sullivan and Tom Igoe, *Physical Computing: Sensing and Controlling the Physical World with Computers* (Thomson Course Technology PTR, 2004), p. 5.

Code

To run these examples, unlike the other examples in this book, you will need additional equipment. They require either a Wiring (*wiring.org.co*) or Arduino (*www.arduino.cc*) board and the following:

USB cable (used to send data between board and computer)
9–15V 1000mA power supply or 9V battery
22-gauge solid core wire (get different colors)
Breadboard
Switch
Resistors (10K ohm for the switch circuits, 330 ohm for the LEDs, 1K ohm for the photoresistor)
LEDs
Servo motor (Futaba or Hi-Tech)
DC motor (a generic DC motor like the ones in toy cars)
L293D or SN754410 H-Bridge Integrated Circuit
Wire cutters
Wire strippers
Needlenose pliers

This equipment can be purchased from an electronics store such as Radio Shack or from an online vendor (p. 658).

Each example presents two programs: code for the I/O board and code for Processing. Diagrams and breadboard illustrations for the examples are side by side on pages 652 and 656 to reinforce the connections between the two representations. Learning to translate a circuit diagram into a physical circuit is one of the most difficult challenges when starting to work with electronics.

The Wiring or Arduino software environment is necessary to program each board. These environments are built on top of the Processing environment, but they have special features for uploading code to the board and monitoring serial communication. Both can be downloaded at no cost from their respective websites and both are available for Linux, Macintosh, and Windows.

The examples that follow assume the I/O board is connected to your computer and serial communication is working. Before working with these examples, get one of the simple Serial library examples included with Processing to work. For the most up-to-date information and troubleshooting tips, read the Serial reference on the Processing website: *www.processing.org/reference/libraries*. The Wiring and Arduino websites have additional information.

Example 1A: Switch (Wiring/Arduino)

```
// Code for sensing a switch status and writing the value to the serial port

int switchPin = 4;                       // Switch connected to pin 4

void setup() {
  pinMode(switchPin, INPUT);             // Set pin 0 as an input
  Serial.begin(9600);                    // Start serial communication at 9600 bps
}

void loop() {
  if (digitalRead(switchPin) == HIGH) {  // If switch is ON,
    Serial.print(1, BYTE);               // send 1 to Processing
  } else {                               // If the switch is not ON,
    Serial.print(0, BYTE);               // send 0 to Processing
  }
  delay(100);                            // Wait 100 milliseconds
}
```

Example 1B: Switch (Processing)

```
// Read data from the serial port and change the color of a rectangle
// when a switch connected to the board is pressed and released

import processing.serial.*;

Serial port;                             // Create object from Serial class
int val;                                 // Data received from the serial port

void setup() {
  size(200, 200);
  frameRate(10);
  // Open the port that the board is connected to and use the same speed (9600 bps)
  port = new Serial(this, 9600);
}

void draw() {
  if (0 < port.available()) {            // If data is available,
    val = port.read();                   // read it and store it in val
  }
  background(255);                       // Set background to white
  if (val == 0)  {                       // If the serial value is 0,
    fill(0);                             // set fill to black
  } else {                               // If the serial value is not 0,
    fill(204);                           // set fill to light gray
  }
  rect(50, 50, 100, 100);
}
```

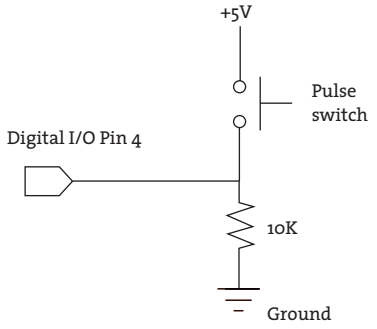
Example 2A: Light sensor (Wiring/Arduino)

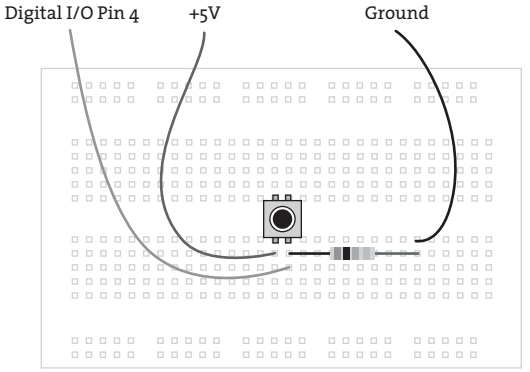
```
// Code to read an analog value and write it to the serial port

int val;
int inputPin = 0;                // Set the input to analog in pin 0

void setup() {
  Serial.begin(9600);            // Start serial communication at 9600 bps
}

void loop() {
  val = analogRead(inputPin)/4;  // Read analog input pin, put in range 0 to 255
  Serial.print(val, BYTE);       // Send the value
  delay(100);                    // Wait 100ms for next reading
}
```

Example 2B: Light sensor (Processing)

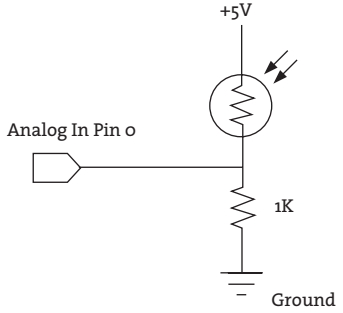
```
// Read data from the serial port and assign it to a variable. Set the fill a
// rectangle on the screen using the value read from a light sensor connected
// to the Wiring or Arduino board

import processing.serial.*;

Serial port;                     // Create object from Serial class
int val;                         // Data received from the serial port

void setup() {
  size(200, 200);
  noStroke();
  frameRate(10);                 // Run 10 frames per second
  // Open the port that the board is connected to and use the same speed (9600 bps)
  port = new Serial(this, 9600);
}

void draw() {
  if (0 < port.available()) {    // If data is available to read,
    val = port.read();           // read it and store it in val
  }
  background(204);               // Clear background
  fill(val);                     // Set fill color with the value read
  rect(50, 50, 100, 100);        // Draw square
}
```

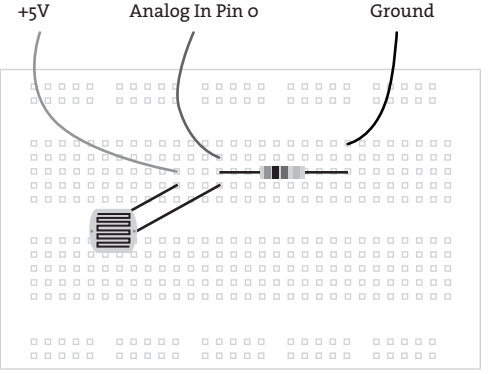
Example 1



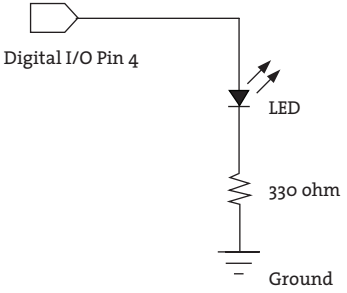


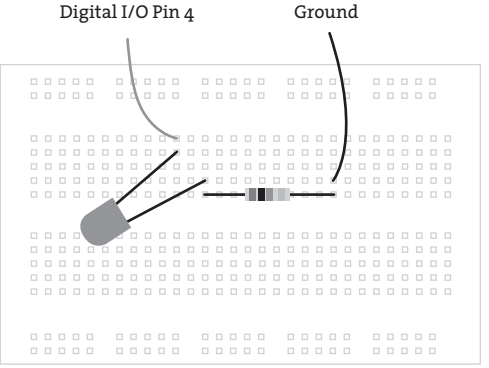
Example 2





Example 3





Example 3A: Turning a light on and off (Wiring/Arduino)

```
// Read data from the serial and turn ON or OFF a light depending on the value

char val;                         // Data received from the serial port
int ledPin = 4;                   // Set the pin to digital I/O 4

void setup() {
  pinMode(ledPin, OUTPUT);        // Set pin as OUTPUT
  Serial.begin(9600);             // Start serial communication at 9600 bps
}

void loop() {
  if (Serial.available()) {       // If data is available to read,
    val = Serial.read();          // read it and store it in val
  }
  if (val == 'H') {               // If H was received
    digitalWrite(ledPin, HIGH);   // turn the LED on
  } else {
    digitalWrite(ledPin, LOW);    // Otherwise turn it OFF
  }
  delay(100);                     // Wait 100 milliseconds for next reading
}
```

Example 3B: Turning a light on and off (Processing)

```
// Check if the mouse is over a rectangle and write the status to the serial port

import processing.serial.*;

Serial port;                      // Create object from Serial class

void setup() {
  size(200, 200);
  noStroke();
  frameRate(10);
  // Open the port that the board is connected to and use the same speed (9600 bps)
  port = new Serial(this, 9600);
}

void draw() {
  background(255);
  if (mouseOverRect() == true)  {  // If mouse is over square,
    fill(204);                    // change color and
    port.write('H');              // send an H to indicate mouse is over square
  } else {                        // If mouse is not over square,
    fill(0);                      // change color and
    port.write('L');              // send an L otherwise
  }
  rect(50, 50, 100, 100);         // Draw a square
}
```

```
boolean mouseOverRect() {        // Test if mouse is over square
  return ((mouseX >= 50) && (mouseX <= 150) && (mouseY >= 50) && (mouseY <= 150));
}
```

Example 4A: Controlling a servomotor (Wiring/Arduino)

```
// Read data from the serial port and set the position of a servomotor
// according to the value

Servo myservo;                   // Create servo object to control a servo
int servoPin = 4;                // Connect yellow servo wire to digital I/O pin 4
int val = 0;                     // Data received from the serial port

void setup() {
  myservo.attach(servoPin);      // Attach the servo to the PWM pin
  Serial.begin(9600);            // Start serial communication at 9600 bps
}

void loop() {
  if (Serial.available()) {      // If data is available to read,
    val = Serial.read();         // read it and store it in val
  }
  myservo.write(val);            // Set the servo position
  delay(15);                     // Wait for the servo to get there
}
```

Example 4B: Controlling a servomotor (Processing)

```
// Write data to the serial port according to the mouseX value

import processing.serial.*;

Serial port;                          // Create object from Serial class
float mx = 0.0;

void setup() {
  size(200, 200);
  noStroke();
  frameRate(10);
  // Open the port that the board is connected to and use the same speed (9600 bps)
  port = new Serial(this, 9600);
}

void draw() {
  background(0);                      // Clear background
  fill(204);                          // Set fill color
  rect(40, height/2-15, 120, 25);     // Draw square

  float dif = mouseX - mx;
  if (abs(dif) > 1.0) {
    mx += dif/4.0;
  }
```

```
  mx = constrain(mx, 50, 149);               // Keeps marker on the screen
  noStroke();
  fill(255);
  rect(50, (height/2)-5, 100, 5);
  fill(204, 102, 0);

  rect(mx-2, height/2-5, 4, 5);              // Draw the position marker
  int angle = int(map(mx, 50, 149, 0, 180)); // Scale the value the range 0-180
  //print(angle + " ");                      // Print the current angle (debug)
  port.write(angle);                         // Write the angle to the serial port
}
```

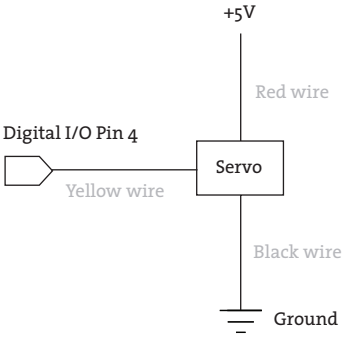
Example 5A: Turning a DC motor on and off (Wiring/Arduino)

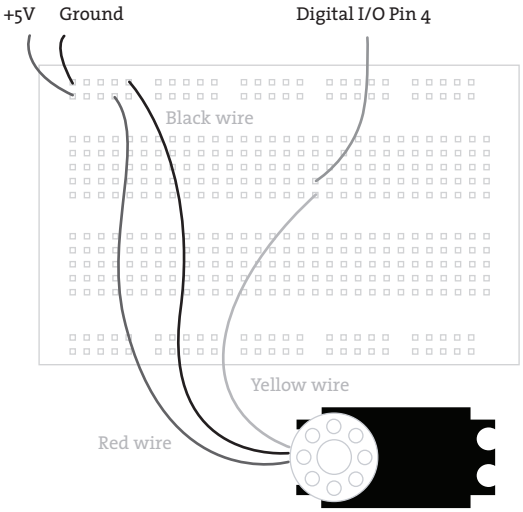
```
// Read data from the serial and turn a DC motor on or off according to the value

char val;                              // Data received from the serial port
int motorpin = 0;     // Wiring: Connect L293D Pin En1 connected to Pin PWM 0
// int motorpin = 9;  // Arduino: Connect L293D Pin En1 to Pin PWM 9

void setup() {
  Serial.begin(9600);                  // Start serial communication at 9600 bps
}

void loop() {
  if (Serial.available()) {            // If data is available,
    val = Serial.read();               // read it and store it in val
  }
  if (val == 'H') {                    // If 'H' was received,
    analogWrite(motorpin, 125);        // turn the motor on at medium speed
  } else {                             // If 'H' was not received
    analogWrite(motorpin, 0);          // turn the motor off
  }
  delay(100);                          // Wait 100 milliseconds for next reading
}
```

Example 5B: Turning a DC motor on and off (Processing)

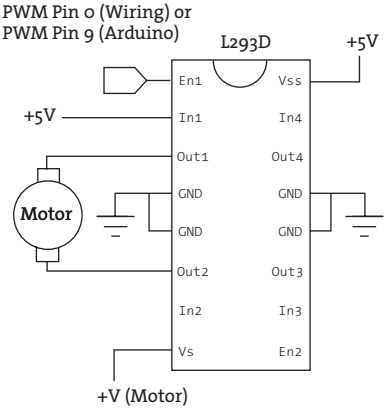
```
// Write data to the serial port according to the status of a button controlled
// by the mouse

import processing.serial.*;

Serial port;                           // Create serial port object
boolean rectOver = false;
int rectX, rectY;                      // Position of square button
int rectSize = 100;                    // Diameter of rect
color rectColor;
boolean buttonOn = false;              // Status of the button
void setup() {
  size(200, 200);
  noStroke();
```

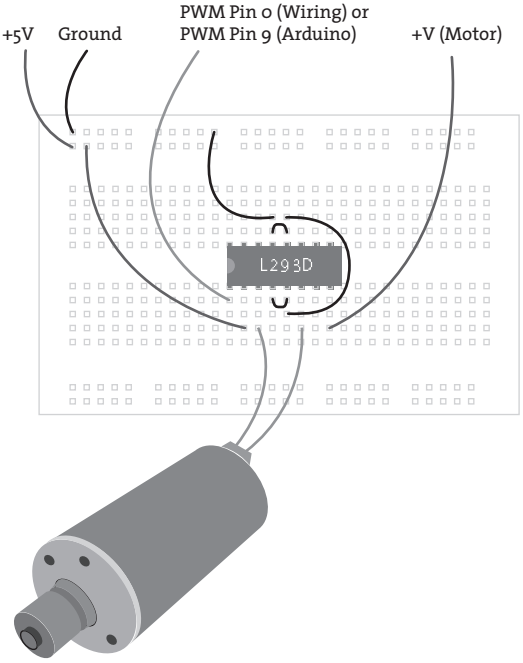
Example 4





Example 5





```
  frameRate(10);
  rectColor = color(100);
  rectX = width/2 - rectSize/2;
  rectY = height/2 - rectSize/2;
  // Open the port that the board is connected to and use the same speed (9600 bps)
  port = new Serial(this, 9600);
}

void draw() {
  update(mouseX, mouseY);
  background(0);                  // Clear background to black
  fill(rectColor);
  rect(rectX, rectY, rectSize, rectSize);
}

void update(int x, int y) {
  if (overRect(rectX, rectY, rectSize, rectSize) == true) {
    rectOver = true;
  } else {
    rectOver = false;
  }
}

void mouseReleased() {
  if (rectOver == true) {
    if (buttonOn == true) {
      rectColor = color(100);
      buttonOn = false;
      port.write('L');           // Send an L to indicate button is OFF
    } else {
      rectColor = color(180);
      buttonOn = true;
      port.write('H');           // Send an H to indicate button is ON
    }
  }
}

boolean overRect(int x, int y, int width, int height) {
  if ((mouseX >= x) && (mouseX <= x+width) &&
      (mouseY >= y) && (mouseY <= y+height)) {
    return true;
  } else {
    return false;
  }
}
```

Resources

Vendors and manufacturers

Acroname. Robotics components distributor and BrainStem creator. http://www.acroname.com.

All Electronics. Surplus electronics distributor. www.allelectronics.com.

Analog Devices. Electronics components (sensors, A/D converters, etc.) manufacturer. http://www.analog.com.

Arduino. Prototyping toolkit. http://www.arduino.cc.

Digi-Key. Large electronics parts distributor. http://www.digikey.com.

Electronic Goldmine. Surplus electronics distributor. http://www.goldmine-elec.com.

EZIO. I/O board designed at the University of Michigan. http://www.ezio.com.

I-Cube X. I/O platform from Infusion Systems. http://infusionsystems.com.

Jameco. Large electronics parts distributor. http://www.jameco.com.

LEGO Mindstorms. Programmable electronics from LEGO. http://mindstorms.lego.com.

McMaster-Carr. Extensive selection of hardware and construction supplies. http://www.mcmaster.com.

Microchip. Creator of PICmicro microcontrollers. http://www.microchip.com.

MicroEngineering Labs. Creator of PicBasic Pro programming environment. http://www.melabs.com.

NetMedia. Creator of BX-24 microcontrollers. http://www.netmedia.com.

Parallax. Creator of the Basic Stamp microcontrollers and related modules. http://www.parallax.com.

Phidgets. I/O platform. http://www.phidgets.com.

Servocity. Servomotor distributor. http://www.servocity.com.

Small Parts. Hardware for researchers and developers. http://www.smallparts.com.

Solarbotics. Distributor for solar-powered robotics kits and components. http://www.solarbotics.com.

Spark Fun Electronics. http://www.sparkfun.com.

Teleo. I/O platform from MakingThings. http://www.makingthings.com.

Wiring. Prototyping environment and toolkit. http://wiring.org.co.

Online resources

Buxton, Bill. List of input devices. http://www.billbuxton.com/InputSources.html.

Igoe, Tom. Physical computing resources. http://tigoe.net/pcomp.

FindChips. Electronic components search engine. http://www.findchips.com.

Instructables. Step-by-step instructions for building projects. http://www.instructables.com.

Haque, Usman, and Adam Somlai-Fischer. Low-tech sensors and actuators. http://lowtech.propositions.org.uk.

Make magazine. Do-it-yourself technology. http://www.makezine.com.

Open Circuits. Wiki for electronics projects, components, and techniques. http://www.opencircuits.com.

O'Sullivan, Dan. Physical computing resources. http://itp.nyu.edu/~dbo3/physical/physical.html.

RS-232 Serial. Data protocol description. http://en.wikipedia.org/wiki/RS-232.

Texts

Burnham, Jack. *Beyond Modern Sculpture*. George Braziller, 1968.

Horowitz, Paul, and Winfield Hill. *The Art of Electronics*. Second edition. Cambridge University Press, 1989.

Hultén, K. G. Pontus. *The Machine: As Seen at the End of the Mechanical Age*. The Museum of Modern Art, 1968.

O'Sullivan, Dan, and Tom Igoe. *Physical Computing: Sensing and Controlling the Physical World with Computers*. Thomson Course Technology PTR, 2004.

Jones, Joseph L., Bruce A. Seiger, and Anita M. Flynn. *Mobile Robots: Inspiration to Implementation*. Second edition. A. K. Peters, 1999.

Fraden, Jacob. *Handbook of Modern Sensors: Physics, Designs, and Applications*. Springer-Verlag, 1996.

Mims, Forrest M. III. *Getting Started in Electronics*. Second edition. Radio Shack, 1998.

Mims, Forrest M. III. *Timer, Op Amp, and Optoelectronic Circuits and Projects*. Radio Shack, 2000.

Petzold, Charles. *Code: The Hidden Language of Computer Hardware and Software*. Microsoft Press, 2000.

Scherz, Paul. *Practical Electronics for Inventors*. McGraw-Hill, 2000.

Wise, Edwin. *Animatrionics: A Guide to Animated Holiday Displays*. PROMPT Publications, 2000.

Artists, designers, institutions

Dunne, Anthony, and Fiona Raby. *Design Noir*. Birkhäuser, 2001.

Bureau of Inverse Technology. http://bureauit.org.

Feingold, Ken. http://www.kenfeingold.com.

Hawkinson, Tim. *Tim Hawkinson*. Harry N. Abrams, 2004.

IDEO. Design firm. http://www.ideo.com.

Interaction Design Institute Ivrea. Former graduate program. http://www.interaction-ivrea.it.

Design Interactions. Royal College of Art. MFA program. http://www.interaction.rca.ac.uk.

Interactive Telecommunication Program. New York University. MFA program. http://itp.nyu.edu/itp.

Jones, Crispin. http://www.mr-jones.org.

Kuwakubo, Ryota. http://www.vector-scan.com.

Maywa Denki. http://www.maywadenki.com.

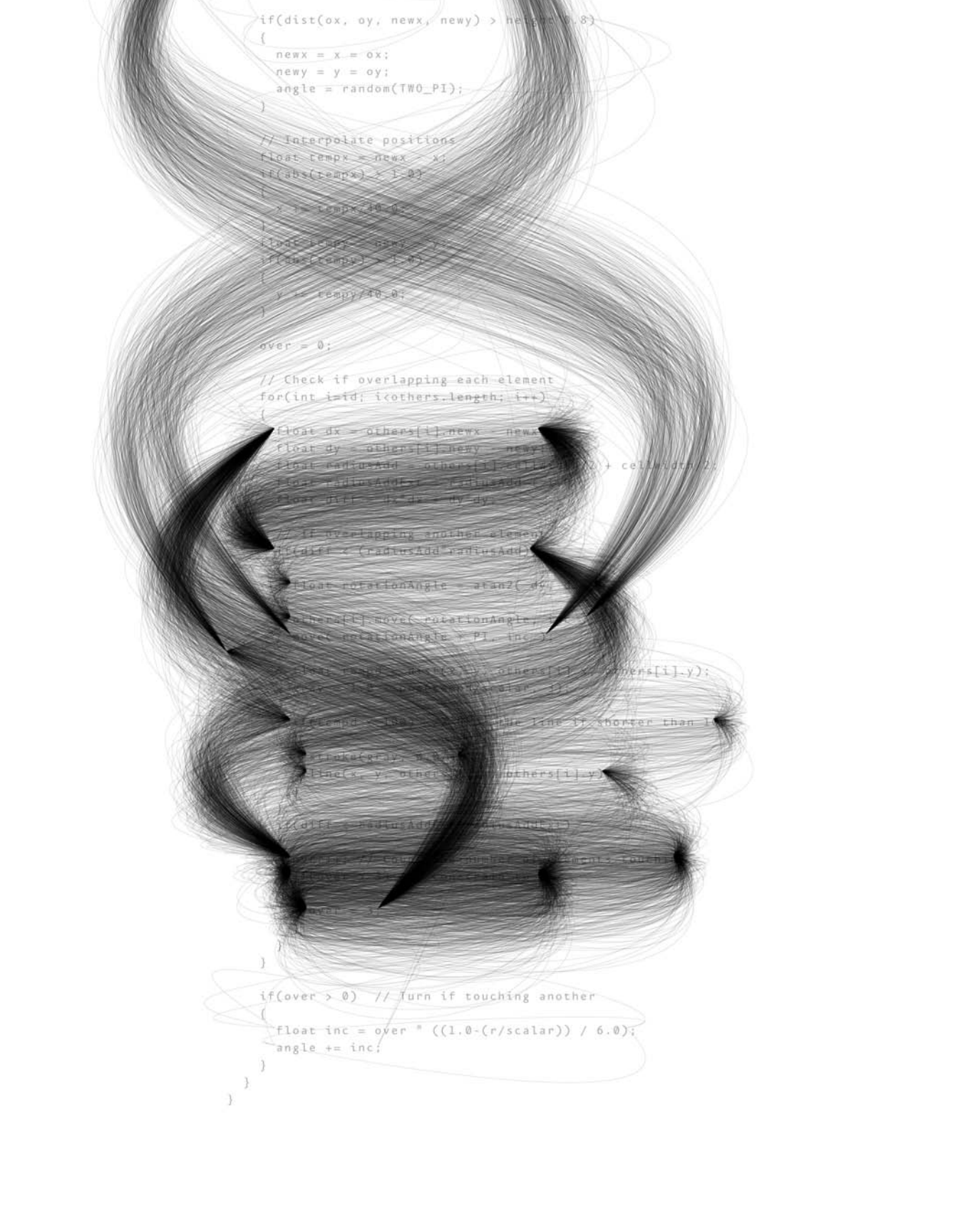
Paik, Nam June. *The Worlds of Nam June Paik*. Solomon R. Guggenheim Foundation, 2000.

Rath, Alan. *Alan Rath: Robotics*. Smart Art Press, 1999.

Rinaldo, Ken. http://kenrinaldo.com.

Tangible Media Group, MIT Media Lab. http://tangible.media.mit.edu.

Vogel, Peter. http://www.bitforms.com/artist_vogel.html.



Appendix A

Order of Operations

An operator is a symbol that performs an operation. There are operators for arithmetic, relational, logical, and bitwise operations. The order of operations determines which operations are performed before others. For example, in the following expression, is the addition or multiplication calculated first?

```
2 + 3 * 4
```

You may remember the mnemonic "My Dear Aunt Sally" (MDAS) from math class. MDAS, short for Multiply, Divide, Add, Subtract, states the order in which mathematical operations in an expression should be performed. Regardless of the order from left to right, the multiplication should happen first (`3 * 4`), and the result should then be added (`2 + 12`). If this order is not followed, the expression will not evaluate to the correct result.

The order of operations goes beyond arithmetic and applies to all of the operators within a programming language. The list of operators presented below reveals the order in which they are evaluated within Processing. The operators higher in this list are evaluated before those on lower lines. This is not a complete list of the operators available in Processing, but it contains those used within this book's examples.

Name	Symbol	Examples
Parentheses	`()`	`a*(b+c)`
Postfix, Unary	`++ -- !`	`a++ --b !c`
Multiplicative	`* / %`	`a * b`
Additive	`+ -`	`a + b`
Relational	`< > <= >=`	`if (a < b)`
Equality	`== !=`	`if (a == b)`
Logical AND	`&&`	`if (mousePressed && (a > b))`
Logical OR	`||`	`if (mousePressed || (a > b))`
Assignment	`= += -= *= /= %=`	`a = 44`

The following examples clarify how the order of operations works:

```
// The expression 4 + 5 evaluates to 9, then the
// value 9 is assigned to the variable x
int x = 4 + 5;
```

A-01

```
// The expression 5 * 10 evaluates to 50, then the
// expression 4 + 50 evaluates to 54, then the
// value 54 is then assigned to the variable x
int x = 4 + 5 * 10;
```

A-02

Ben Fry. *Disarticulate*, 2006.

In the previous example, even though * is the last operator on the line, it is evaluated first because the multiplicative operators have precedence over the additive and assignment operators. Parentheses are used to control the evaluation of an expression if a different order of operation is desired. In this example, they are used to evaluate the addition before the multiplication:

A-03

```
// The expression 4 + 5 evaluates to 9, then the
// expression 9 * 10 evaluates to 90, then the
// value 90 is assigned to the variable x
int x = (4 + 5) * 10;
```

When operators with the same evaluation order (e.g., + and -, * and /) appear within an expression, they are evaluated from left to right. The following example demonstrates this rule:

A-04

```
float w = 12.0 - 6.0 + 3.0;  // Assigns 9 to w
float x = 3.0 + 6.0 / 12.0;  // Assigns 3.5 to x
float y = 12.0 / 6.0 * 3.0;  // Assigns 6 to y
float z = 3.0 * 6.0 / 12.0;  // Assigns 1.5 to z
```

When in doubt, it is helpful to include parentheses to specify the desired order of operations. This can be a useful reminder when returning to the code after some time, and it can help clarify the intent for others reading the code.

Appendix B

Reserved Words

Some words are essential to the Processing language, and their use is restricted to their intended use. For example, the `int` data type is an integral component of the language. It can be used only to declare a variable of that type and cannot be used as the name for a variable. The following program produces an error:

```
float int = 50;  // ERROR! Unexpected token: float
line(int, 0, int, 100);
```

B-01

Many of these language components are inherited from the Java programming language (Processing is built using Java), but some are unique to Processing. The names of the functions, variables, and constants such as `line`, `fill`, `width`, `mouseX`, and `PI` found in the Processing reference should be used only as intended. For example, while it's possible to make a variable called `line`, it can make a program confusing to read. When someone sees the word `line` in a program, there is an expectation that it will be used to run the `line()` function. The following program demonstrates how ignoring this suggestion can make a program baffling:

```
int line = 50;              // This does not create a program error
line(line, 0, line, 100);   // but it's very confusing
```

B-02

In addition to the words used for variables and functions in the Processing language, the list below presents words that cannot be used as names for functions, variables, and classes within a Processing program. Most of these will cause an error that may be quite hard to identify because of the way it confuses the compiler.

Processing's reserved words

abstract	do	implements	protected	this
assert	double	import	public	throw
boolean	else	init	return	throws
break	enum	instanceof	setup	transient
byte	extends	int	short	true
case	false	interface	start	try
catch	final	long	static	update
char	finally	native	stop	void
class	float	new	strictfp	volatile
const	for	null	super	while
continue	goto	package	switch	
default	if	private	synchronized	

Appendix C

ASCII, Unicode

ASCII (pronounced AS-kee) is an acronym for American Standard Code for Information Interchange. It was designed as a code to associate the letters, numbers, and punctuation in the English language with numeric values. Characters must be translated into numbers so they can be used as data in software, and ASCII is a standard for making this conversion. Processing distinguishes between the number and character representation of a value based on the data type. For example, the binary sequence 01000001 will be used as the character *A* if the data type is `char` and will be used as the number 65 if the data type is `int`.

The ASCII standard also has encodings for control characters (nonprintable characters) such as tab, backspace, enter, escape, line feed, etc., which can control external devices such as printers and format whitespace in a file. Control characters are important when writing and reading files and for reading keys such as Tab, Del, and Esc on the keyboard.

ASCII became a standard in 1967 and was last updated in 1986. Over time it has been replaced by newer standards such as UTF-8 and Unicode, which define a wider range of characters and therefore can be used for text written in non-English languages. ASCII remains extremely useful because of its simplicity and ubiquity. More information about ASCII can be found at *http://en.wikipedia.org/wiki/ASCII*.

The tables on the opposite page present the relations between the various representations of each character. Additional control characters can be seen in a full ASCII chart but are omitted here for brevity.

Because of its heritage, each ASCII character consists of only seven bits, covering the numbers 0 through 127. Because a byte is 8 bits and can express numbers between 0 and 255, the values from 128 to 255 vary widely between operating systems and locales. Because each platform interprets values between 128 and 255 differently, a plain-text file that contains a *ü* character that is created on Mac OS will be interpreted incorrectly on a Windows machine unless the software is told that the file is using a Macintosh encoding. It's common to see Web pages that contain strange characters instead of a slanted double quote or an em dash. This is the result of a Web page that was created on a Macintosh being viewed on a PC (or vice versa). To prevent this situation, the HTML specification has its own method of encoding characters as plain ASCII, even though many don't use it. A table that translates between each of these encodings appears below.

The Unicode standard (*www.unicode.org*) is an attempt to create a universal character set that encompasses many international languages. Most commonly, Unicode characters are represented as two bytes, which means that 65,536 characters can be specified. A `char` in Processing is a single Unicode character. Unicode information is often stored in a format called UTF-8, which uses a clever packing mechanism to store the 16-bit values as mostly 8-bit data.

ASCII Characters (Numeric value followed by the corresponding character)

32	(space)	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	“	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	‘	58	:	77	M	96	`	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

ASCII Control Characters (Abridged)

Number	Abbreviation	Escape Sequence	Processing Constant	Description
0	NUL			Null character
4	EOT (EOF)			End of transmission (or file)
6	ACK			Acknowledgment
7	BEL			Bell
8	BS	\b	BACKSPACE	Backspace
9	HT	\t	TAB	Horizontal tab
10	LF	\n	ENTER	Line feed
13	CR	\r	RETURN	Carriage return
27	ESC		ESC	Escape
127	DEL		DELETE	Delete

When including non-ASCII characters in a Processing program, it's a good idea to use each character's *escape sequence* (p. 421), rather than the actual character, so that problems aren't introduced when the file is opened on another platform (e.g., source code for a sketch is sent by email or posted as a text file on the Web). The escape sequence consists of `\u` followed by four digits that represent the character's Unicode value in hexadecimal. The chart below also includes the escape sequences for many characters. For instance, instead of the following . . .

```
text("Zoë", 50, 50);
```

. . . this is more compatible and therefore safer:

```
text("Zo\u00EB", 50, 50);
```

Some fonts, such as Arial Unicode, support thousands of characters in a single font. When the Create Font tool is used with such a font, it's possible to include all available characters by selecting the "All characters" option. As might be expected, this can produce enormous files and may even cause an `OutOfMemoryError`. Without the "All characters" option selected, fonts are encoded with all ASCII values, plus the characters found in the table below. This table is based on the most common Roman-language encodings for Mac OS and Windows (Mac Roman and CP1250), plus ISO Latin-1, a third encoding that defines the Unicode characters 128 through 255.

Character Format Conversion Table for the Processing Character Set

Character	Processing Escape	Unicode Decimal	Mac Roman	Windows CP1250	HTML Escape
‚	\u0082	130			‚
ƒ	\u0083	131			ƒ
„	\u0084	132			„
…	\u0085	133			…
†	\u0086	134			†
‡	\u0087	135			‡
ˆ	\u0088	136			ˆ
‰	\u0089	137			‰
Š	\u008A	138			Š
‹	\u008B	139			‹
Œ	\u008C	140			Œ
[RI]	\u008D	141			
Ž	\u008E	142			Ž
[SS3]	\u008F	143			
[DCS]	\u0090	144			
‘	\u0091	145			‘
’	\u0092	146			’
“	\u0093	147			“
”	\u0094	148			”
•	\u0095	149			•
–	\u0096	150			–
—	\u0097	151			—
˜	\u0098	152			˜
™	\u0099	153			™
š	\u009A	154			š
›	\u009B	155			›
œ	\u009C	156			œ
[OSC]	\u009D	157			
ž	\u009E	158			ž
Ÿ	\u009F	159			Ÿ
[NBSP]	\u00A0	160	202	160	
¡	\u00A1	161	193		¡
¢	\u00A2	162	162		¢
£	\u00A3	163	163		£
¤	\u00A4	164	219	164	¤
¥	\u00A5	165	180		¥
¦	\u00A6	166		166	¦
§	\u00A7	167	164	167	§
¨	\u00A8	168	172	168	¨
©	\u00A9	169	169	169	©
ª	\u00AA	170	187		ª
«	\u00AB	171	199	171	«
¬	\u00AC	172	194	172	¬
[SHY]	\u00AD	173		173	
®	\u00AE	174	168	174	®
¯	\u00AF	175	248		¯

°	\u00B0	176	161	176	°
±	\u00B1	177	177	177	±
²	\u00B2	178			²
³	\u00B3	179			³
´	\u00B4	180	171	180	´
µ	\u00B5	181	181	181	µ
¶	\u00B6	182	166	182	¶
·	\u00B7	183	225	183	·
¸	\u00B8	184	252	184	¸
¹	\u00B9	185			¹
º	\u00BA	186	188		º
»	\u00BB	187	200	187	»
¼	\u00BC	188			¼
½	\u00BD	189			½
¾	\u00BE	190			¾
¿	\u00BF	191	192		¿
À	\u00C0	192	203		À
Á	\u00C1	193	231	193	Á
Â	\u00C2	194	229	194	Â
Ã	\u00C3	195	204		Ã
Ä	\u00C4	196	128	196	Ä
Å	\u00C5	197	129		Å
Æ	\u00C6	198	174		Æ
Ç	\u00C7	199	130	199	Ç
È	\u00C8	200	233		È
É	\u00C9	201	131	201	É
Ê	\u00CA	202	230		Ê
Ë	\u00CB	203	232	203	Ë
Ì	\u00CC	204	237		Ì
Í	\u00CD	205	234	205	Í
Î	\u00CE	206	235	206	Î
Ï	\u00CF	207	236		Ï
Ð	\u00D0	208			Ð
Ñ	\u00D1	209	132		Ñ
Ò	\u00D2	210	241		Ò
Ó	\u00D3	211	238	211	Ó
Ô	\u00D4	212	239	212	Ô
Õ	\u00D5	213	205		Õ
Ö	\u00D6	214	133	214	Ö
×	\u00D7	215		215	×
Ø	\u00D8	216	175		Ø
Ù	\u00D9	217	244		Ù
Ú	\u00DA	218	242	218	Ú
Û	\u00DB	219	243		Û
Ü	\u00DC	220	134	220	Ü
Ý	\u00DD	221		221	Ý
Þ	\u00DE	222			Þ
ß	\u00DF	223	167	223	ß
à	\u00E0	224	136		à
á	\u00E1	225	135	225	á
â	\u00E2	226	137	226	â
ã	\u00E3	227	139		ã
ä	\u00E4	228	138	228	ä
å	\u00E5	229	140		å
æ	\u00E6	230	190		æ
ç	\u00E7	231	141	231	ç
è	\u00E8	232	143		è
é	\u00E9	233	142	233	é
ê	\u00EA	234	144		ê
ë	\u00EB	235	145	235	ë
ì	\u00EC	236	147		ì
í	\u00ED	237	146	237	í
î	\u00EE	238	148	238	î
ï	\u00EF	239	149		ï
ð	\u00F0	240			ð
ñ	\u00F1	241	150		ñ
ò	\u00F2	242	152		ò
ó	\u00F3	243	151	243	ó
ô	\u00F4	244	153	244	ô
õ	\u00F5	245	155		õ
ö	\u00F6	246	154	246	ö
÷	\u00F7	247	214	247	÷
ø	\u00F8	248	191		ø
ù	\u00F9	249	157		ù
ú	\u00FA	250	156	250	ú
û	\u00FB	251	158		û
ü	\u00FC	252	159	252	ü
ý	\u00FD	253		253	ý
þ	\u00FE	254			þ
ÿ	\u00FF	255	216		ÿ
Ă	\u0102	258		195	Ă
ă	\u0103	259		227	ă
Ą	\u0104	260		165	Ą
ą	\u0105	261		185	ą
Ć	\u0106	262		198	Ć
ć	\u0107	263		230	ć
Č	\u010C	268		200	Č
č	\u010D	269		232	č
Ď	\u010E	270		207	Ď
ď	\u010F	271		239	ď
Đ	\u0110	272		208	Đ
đ	\u0111	273		240	đ
Ę	\u0118	280		202	Ę
ę	\u0119	281		234	ę
Ě	\u011A	282		204	Ě
ě	\u011B	283		236	ě
ı	\u0131	305	245		ı
Ĺ	\u0139	313		197	Ĺ
Í	\u013A	314		229	ĺ
Ľ	\u013D	317		188	Ľ
ľ	\u013E	318		190	ľ
Ł	\u0141	321		163	Ł
ł	\u0142	322		179	ł
Ń	\u0143	323		209	Ń

ń	\u0144	324		241	ń
Ň	\u0147	327		210	Ň
ň	\u0148	328		242	ň
Ő	\u0150	336		213	Ő
ő	\u0151	337		245	ő
Œ	\u0152	338	206		Œ
œ	\u0153	339	207		œ
Ŕ	\u0154	340		192	Ŕ
ŕ	\u0155	341		224	ŕ
Ř	\u0158	344		216	Ř
ř	\u0159	345		248	ř
Ś	\u015A	346		140	Ś
ś	\u015B	347		156	ś
Ş	\u015E	350		170	Ş
ş	\u015F	351		186	ş
Š	\u0160	352		138	Š
š	\u0161	353		154	š
Ţ	\u0162	354		222	Ţ
ţ	\u0163	355		254	ţ
Ť	\u0164	356		141	Ť
ť	\u0165	357		157	ť
Ů	\u016E	366		217	Ů
ů	\u016F	367		249	ů
Ű	\u0170	368		219	Ű
ű	\u0171	369		251	ű
Ÿ	\u0178	376	217		Ÿ
Ź	\u0179	377		143	Ź
ź	\u017A	378		159	ź
Ż	\u017B	379		175	Ż
ż	\u017C	380		191	ż
Ž	\u017D	381		142	Ž
ž	\u017E	382		158	ž
ƒ	\u0192	402	196		ƒ
ˆ	\u02C6	710	246		ˆ
ˇ	\u02C7	711	255	161	ˇ
˘	\u02D8	728	249	162	˘
˙	\u02D9	729	250	255	˙
°	\u02DA	730	251		˚
˛	\u02DB	731	254	178	˛
˜	\u02DC	732	247		˜
˝	\u02DD	733	253	189	˝
Ω	\u03A9	937	189		Ω
π	\u03C0	960	185		π
–	\u2013	8211	208	150	–
—	\u2014	8212	209	151	—
‘	\u2018	8216	212	145	‘
’	\u2019	8217	213	146	’
‚	\u201A	8218	226	130	‚
“	\u201C	8220	210	147	“
”	\u201D	8221	211	148	”
„	\u201E	8222	227	132	„
†	\u2020	8224	160	134	†
‡	\u2021	8225	224	135	‡
•	\u2022	8226	165	149	•
…	\u2026	8230	201	133	…
‰	\u2030	8240	228	137	‰
‹	\u2039	8249	220	139	‹
›	\u203A	8250	221	155	›
⁄	\u2044	8260	218		⁄
€	\u20AC	8364		128	€
™	\u2122	8482	170	153	™
∂	\u2202	8706	182		∂
Δ	\u2206	8710	198		∆
∏	\u220F	8719	184		∏
∑	\u2211	8721	183		∑
√	\u221A	8730	195		√
∞	\u221E	8734	176		∞
∫	\u222B	8747	186		∫
≈	\u2248	8776	197		≈
≠	\u2260	8800	173		≠
≤	\u2264	8804	178		≤
≥	\u2265	8805	179		≥
◊	\u25CA	9674	215		◊
	\uF8FF	63743	240		
fi	\uFB01	64257	222		ﬁ
fl	\uFB02	64258	223		ﬂ

Appendix D

Bit, Binary, Hex

Bit

A bit (binary digit) is the most basic information unit in computing. It's often thought of as a 1 or 0, but a bit has no numeric meaning. It's simply a way to distinguish between two mutually exclusive states. Bits may be stored as holes punched in a card, a positive or negative magnetic charge on a floppy disk, or an indent in the surface of a compact disk. The amazing innovation of binary notation is the ability to encode many types of data and logic with only two different states. This was made possible by George Boole's contributions to logic in the mid-nineteenth century and Claude Shannon's development of information theory in the 1930s. The information that comprises images, video, text, and software is all encoded into binary notation and later decoded as colors, shapes, and words that we are able to understand. Bits are grouped together in units of 8 called bytes. Storage on computers is measured in these units. For example, a kilobyte (K, KB, kB, Kbyte) is 1024 bytes, a megabyte (MB) is 1,048,576 bytes, and a gigabyte (GB, Gbyte) is 1,073,741,824 bytes.[1]

Binary

The binary number system, also called base-2, represents numbers as sequences of 1s and 0s. This is different from the more common decimal representation, also called base-10. Here we can compare the powers of 10 and the powers of 2:

Base-10	10^0	10^1	10^2	10^3	10^4	10^5
	1	10	100	1000	10000	100000
Base-2	2^0	2^1	2^2	2^3	2^4	2^5
	1	2	4	8	16	32

When using a computer, it's clear that many frequently used numbers are a result of base-2 notation. For example, colors are specified in values from 0 to 255 (2^8, the number of unique values for one byte), and screens are often 1024 pixels wide (2^{10}).

In base-10 numbers, each digit is multiplied by the power of 10 associated with its position. For example, the number 243 is expressed as follows:

$200 + 40 + 3 = 243$
$2*100 + 4*10 + 3*1 = 243$
$2*10^2 + 4*10^1 + 3*10^0 = 243$

Base-2 numbers work the same way, but the digits are multiplied by the powers of 2. Every whole number can be made by adding values that are powers of two. The following example breaks down the binary equivalent of the decimal number 23 (16+4+2+1) which is 10111:

```
16     +  0     +  4     +  2     +  1     =  23
1*16   +  0*8   +  1*4   +  1*2   +  1*1   =  23
1*2^5  +  0*2^4 +  1*2^2 +  1*2^1 +  1*2^0 =  23
```

Each file format specifies how information is encoded into binary notation and how software is used to decode the information according to the standards for each format. For example, using the ASCII standard (p. 664) for text, the word Process is encoded as the numeric and binary sequence like this:

Character	P	r	o	c	e	s	s
ASCII	80	114	111	99	101	115	115
Binary	01010000	01110010	01101111	01100011	01100101	01110011	01110011

Bitwise operations

The integer and floating-point representations of numbers are operated on with arithmetic operators such as + and *. The binary representations of numbers have different operators. Bitwise operators & (bitwise AND) and | (bitwise OR) are used for comparing binary representations. The bitwise operators >> and << are used to shift bits left and right.

The bitwise AND operator compares each corresponding bit according to these rules:

Expression	Evaluation
1 & 1	1
1 & 0	0
0 & 0	0

A larger calculation follows:

```
  11010110
& 01011100
  --------
  01010100
```

The bitwise OR operator compares each corresponding bit according to these rules.

Expression	Evaluation
1 | 1	1
1 | 0	1
0 | 0	0

A larger calculation follows:

```
  11010110
| 01011100
  ────────
  11011110
```

The bitwise operators >> and << shift bits left and right.

```
int a = 205;  // In binary:  00000000000000000000000011001101
int b = 45;   // In binary:  00000000000000000000000000101101
a = a << 24;  // Converts to 11001101000000000000000000000000
b = b << 8;   // Converts to 00000000000000000010110100000000
```

D-01

Hex

Hexadecimal notation encodes an entire 8-digit byte with just two characters, one character for each *nibble* (4 bits, or half a byte). Because there are only 16 possible byte configurations for a nibble, each can be encoded with the following 16 distinct alphanumeric characters.

0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

For example, the binary sequence . . .

```
01010000 01110010 01101111 01100011 01100101 01110011 01110011
```

. . . is reduced to this hexadecimal encoding:

```
50 72 6F 63 65 73 73
```

Hex notation is an established way to define color within software and on the Web. For example, the three decimal RGB color value 255, 204, 51 is converted to FFCC33 in

hexadecimal notation. In Processing, a # sign is front of six digits denotes a web color. To use hexadecimal notation for other uses besides color, place `0x` (the number zero followed by a lowercase x) in front of the digits.

Notes

1. There are two definitions each for kilobyte, megabyte, and gigabyte. The alternative quantities are, respectively, 1,000 (one thousand), 1,000,000 (one million), and 1,000,000,000 (one billion).

Appendix E

Optimization

Optimization is making changes to a program so that it will run faster. This can provide tremendous benefit by increasing the number of frames displayed per second or by allowing more to be drawn to the screen each frame. Increasing the speed can also make a program more responsive to mouse and keyboard input.

Code should usually not be optimized until a late stage in a program's development. Energy diverted to optimization detracts from refining the concept and aesthetic considerations of the software. Optimization can be very rewarding because of increased performance, but such technical details should not be allowed to distract you from the ideas. There are a few important heuristics to guide the process:

Work slowly and carefully. It's easy to introduce new bugs when optimizing, so work with small pieces of code at a time. Always keep the original version of the code. You may want to comment out the old version of a function and keep it present in your program while you work on its optimization.

Optimize the code that's used most. The majority of code in a program gets used very little, so make sure that you're focusing on a section that needs work. Advanced programmers can use a *profiler* for this task—a tool that identifies the amount of time being spent in different sections of code. Profilers are too specific to be covered here, but books and online references cover profiling Java code, and this methodology can be applied to Processing programs.

If the optimization doesn't help, revert to the original code. Lots of things seem like they'll improve performance but actually don't (or they don't improve things as much as hoped). The "optimized" version of the code will usually be more difficult to read—so if the benefits aren't sufficient, the clearer version is better.

There are many techniques to optimize programs; some that are particularly relevant to Processing sketches are listed below.

Bit-shifting color data

The `red()`, `green()`, `blue()`, and `alpha()` functions are easy to use and understand, but because they take the `colorMode()` setting into account, using them is much slower than making direct operations on the numbers. With the default color mode, the same numerical results can be achieved with greater speed by using the `>>` (right shift) operator to isolate the components and then use the bit mask `0xFF` to remove any unwanted data. These operators are explained in Appendix D (p. 669). The following example shows how to shift color data to isolate each component.

E-01

```
color c = color(204, 153, 102, 255);
float r = (c >> 16) & 0xFF;  // Faster version of red(c)
float g = (c >> 8) & 0xFF;   // Faster version of green(c)
float b = c & 0xFF;          // Faster version of blue(c)
float a = (c >> 24) & 0xFF;  // Faster version of alpha(c)
println(r + ", " + g + ", " + b + ", " + a);
```

Each component of a color is 8 bits. The bits in a pixel are ordered like this . . .

```
AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB
```

. . . where the A's are alpha bits, R's are red, G's are green, and B's are blue. After the red component is shifted 16 to the right (the `>> 16` above), the pixel values looks like this:

```
00000000000000000AAAAAAAARRRRRRRR
```

The hexadecimal value `0xFF` is 8 bits all set to 1 (equivalent to 255). The bitwise AND operator is applied to remove the other bits and just save the 8 bits needed for the color component itself:

```
  00000000000000000AAAAAAAARRRRRRRR
& 00000000000000000000000011111111
----------------------------------
  00000000000000000000000RRRRRRRR
```

This calculation completely isolates the red component of the color. To put a color "back together," use the following code:

E-02

```
int a = 255;
int r = 102;
int g = 51;
int b = 255;
color c = (a << 24) | (r << 16) | (g << 8) | b;
```

The following code is useful when the alpha setting is opaque, because no shift is needed:

```
color c = 0xFF000000 | (r << 16) | (g << 8) | b;
```

The hex digits 0xFF000000 (p. 671) are equivalent to 255 shifted to the left by 24. Hex digits can be used to replace the red and green components as well.

Bit shifting is much faster than using the `color()` method, because it ignores the `colorMode()` setting. For the same reason, specifying colors using hex notation (e.g., `#FFCC00`) has zero overhead.

Avoid creating objects in `draw()`

Creating an object slows a program down. When possible, create the objects within `setup()` so they are created only once within the program. For example, load all images and create objects within `setup()`. The following two examples show common misunderstandings about creating objects that slow programs down.

```
// AVOID loading an image within draw(); it is slow                    E-03
void draw()  {
  PImage img = loadImage("tower.jpg");
  image(img, 0, 0);
}
```

```
// AVOID creating an array inside draw(); it is slow                   E-04
void draw()  {
  int[] values = new int[200];
  // Do something with the array here
}
```

In this case, the array will be re-created and destroyed on each trip through the `draw()` method, which is extremely wasteful. The programs 43-07 (p. 404) and 44-05 (p. 417) show faster ways of creating objects.

Using the `pixels[]` array

The `get()` and `set()` functions are easy to use, but they are not as fast as accessing and setting the pixels of an image directly through the `pixels[]` array (p. 356). The following examples show four different ways of accessing the data within the `pixels[]` array, each faster than the previous one.

```
// Converts (x,y) coordinates into a position in the pixels[] array    E-05
loadPixels();
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    pixels[y*height + x] = color(102);
  }
}
updatePixels();
```

```
// Replaces the multiplication y*height with an addition               E-06
int offset = 0;
loadPixels();
for (int y = 0; y < height; y++) {
```

```
  for (int x = 0; x < width; x++) {                    E-06
    pixels[offset + x] = color(102);                   cont.
  }
  offset += width; // Avoids the multiply
}
updatePixels();
```

```
// Avoid the calculation y*height+width                E-07
int index = 0;
loadPixels();
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    pixels[index++] = color(102);
  }
}
updatePixels();
```

```
// Avoids (x,y) coordinates                            E-08
int wh = width*height;
loadPixels();
for (int index = 0; index < wh; index++) {
    pixels[index] = color(102);
}
updatePixels();
```

```
// Only calculate the color once                       E-09
int wh = width*height;
color c = color(102);
loadPixels();
for (int index = 0; index < wh; index++) {
    pixels[index] = c;
}
updatePixels();
```

When manipulating `pixels[]`, use the `loadPixels()` and `updatePixels()` functions only once within `draw()`. When possible, use `color()` outside of loops. It is not very fast because it must take into account the current color mode.

Tips for working with arrays

Adding one value at a time to an array is slower than doubling the size of the array when it's full. If a program will be continually adding data to the end of an array, use the `expand()` function once each time the array fills up in place of running `append()`

many times. Code 33-19 (p. 309) shows how to manage a growing array with `expand()`.

The `arraycopy()` function is the fastest way of copying data from one array to another. Copying the data from one array to another inside a `for` structure is much slower when copying data from large arrays. The `arraycopy()` function is demonstrated in code 33-20 (p. 310). Arrays are also much faster (sometimes 2×) than the Java classes ArrayList and Vector.

Avoid repeating calculations

If the same calculation is made more than once, it's faster to make the calculation once and store it in a variable. Instead of writing . . .

```
float x = (width/2) * 4;
float y = (width/2) * 8;
```

. . . save the result of the division in a variable and substitute it for the calculation:

```
float half = width/2;
float x = half * 4;
float y = half * 8;
```

Multiplications and divisions from inside a `for` structure can slow a program down significantly, especially if there are more than 10,000 iterations. When possible, make these calculations outside of the structure. This is demonstrated above in code E-08.

Because of the way computers make calculations, addition is faster than multiplication and multiplication is faster than division. Multiplication can often be converted to addition by restructuring the program. For example, compare the difference in run time between code E-05 and code E-06.

Lookup tables

The idea behind a lookup table is that it is faster to make reference to a value stored within a data structure than to make the calculation. One example is making calculations for `sin()` and `cos()` at each frame. These numbers can be generated once within `setup()` and stored within an array so they may be quickly retrieved. The following example shows how it's done.

E-10

```
int res = 16;                          // Number of data elements
float[] x = new float[res];            // Create x-coordinate array
float[] y = new float[res];            // Create y-coordinate array

void setup() {
  size(100, 100);
```

E-10 cont.

```
  for (int i = 0; i < res; i++) {
    x[i] = cos(PI/res * i);             // Sets x-coordinates
    y[i] = sin(PI/res * i);             // Sets y-coordinates
  }
}

void draw() {
  for (int i = 0; i < res; i++) {       // Access each point
    point(50 + x[i]*40, 50 + y[i]*40);  // Draws point on a curve
  }
}
```

You can change the resolution of the values by altering the length of the array.

Optimizers beware!

Optimized code can sometimes be more difficult to read. When deciding whether to optimize, balance the need for speed against the value of legibility. For example, in the bit-shifting example presented above, the expression

```
red(color)
```

is more clear than its optimized equivalent:

```
(color >> 16) & 0xFF
```

The name of the `red()` function clearly states its purpose, whereas the bit shift and mask are cryptic to everyone except those familiar with the technique. The confusion can be alleviated with comments, but always consider whether the optimization is more important than the simplicity of your code.

Appendix F

Programming Languages

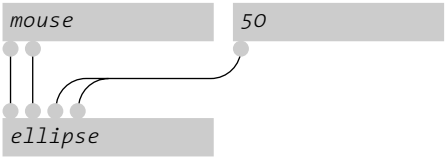
There are hundreds of different programming languages,[1] and many are used by artists and designers. Like human languages, programming languages can be grouped into related sets. French and Spanish are similar because they share similar origins, and the Java and C++ programming languages are similar because they too share similar origins. The etymology of a programming language determines many of its characteristics. Is it appropriate for sketching? Can it be used to program a mobile phone? Is the code highly optimized? Programming languages evolve over a long period of time, and a new language frequently adopts conventions from its predecessors. This appendix describes the characteristics of a variety of languages and their role as tools for artists and designers.

Text vs. visual languages

The first programming languages were text languages, and the majority of programming languages used in the twenty-first century are text languages. Visual programming languages (VPL or VL) are used less often, but they may be employed by a higher percentage of people involved in the arts than in other areas. VPLs often appeal to people who think spatially and prefer to organize their thoughts as visual relationships between elements. For example, the following short text program . . .

```
ellipse(mouseX, mouseY, 50, 50);
```

. . . might be written in a visual programming language like this:



Because VPLs represent software less abstractly, they have proved more effective within specific domains than as general-purpose programming languages. They have found a niche in applications for generating sound, editing video, and building GUIs. A VPL places a greater distance between the programmer and the low-level technical details of the software. For example, a color can be selected directly and curves can be modified and drawn with the mouse rather than specified with numbers. VPLs are themselves written using general-purpose text languages such as C++ and Java; therefore, adding intrinsic features to a VPL requires text programming.

Some researchers feel VPLs have the power to reach an audience that has previously not been attracted to programming. There is logic in this hypothesis, considering that the introduction of the GUI in the 1980s brought the personal computer to a vast new audience. The Scratch project,[2] developed by Mitchel Resnick's research group at the MIT Media Laboratory, was created to enable children to make their own games and animated stories. Programs are created by snapping together visual blocks and setting parameters. There are different block types for mouse events, iteration, and other programming fundamentals. The software is used in the Computer Clubhouse network of after-school centers in low-income communities.

One category of languages is not intrinsically better than the other. Each should be evaluated, and the selection should be made in relation to the programming context and the preferences of the programmer.

Compiled vs. interpreted languages

A program written in a compiled language must be converted into a different format before it is run. The program goes through a process to change it from its human-readable text format into a machine-readable format. This is called *compiling* the program. A program called a *compiler* makes this transition. The program is converted from the representation created by the programmer to a reduced set of instructions (machine code) that can be executed by the computer's microprocessor. A program written in an interpreted language is not compiled—it is *interpreted* by another program while it runs. An *interpreter* is a program that analyzes each statement in the program while it runs and determines what to do. In contrast, all of the necessary decisions about a compiled program are made during the compilation process.

Both types of programming techniques have their strengths. For example, compiled programs run faster than interpreted programs, but interpreted programs can be modified while the program is running. This makes interpreted programs ideal for writing live performance software. Because each language type has advantages, large software projects often utilize both, for different parts of the project where each strength is needed.

The Java language system has aspects of both a compiled and an interpreted language. Before a Java program is run, it's compiled into byte code that is run on the Java Virtual Machine (JVM). The JVM is a software processor that acts as a buffer between the byte code and the machine's physical microprocessor. Because there is a standardized JVM for many different types of computers, the same Java code can theoretically run on all of these different machines without requiring platform-specific changes. The byte code technique makes it easier for code to be readable across platforms without the speed reduction of an interpreted language.

Many interpreted languages are categorized as *scripting languages*. There is no clear definition of a scripting language, but there are properties commonly used to identify one. They are typically created for specific domains or applications. For example, MEL was developed for Maya, ActionScript for Flash, and JavaScript for the Web. A scripting

language provides easy access to specific tasks relevant to a particular context. For example, AppleScript for Mac OS can be used to process a folder full of images and then upload them to a server or send them via Email. Programs can usually be written more quickly in a scripting language, but they often run slower and use more of the computer's memory. It can take less time to write programs in a scripting language because it does not require the programmer to be as specific. For example, scripting languages are often not typed, meaning the data types for variables need not be declared. There's a stereotype that scripting languages are useful only for writing short, simple programs. While they are good for this purpose, scripting languages such as Perl and Python are frequently used to create long, complex programs.

Java was chosen as the basis for Processing because it represented a good balance between performance and simplicity of use. If we didn't care about speed, a scripting language like Python or Ruby might make more sense in a sketching environment. If we didn't care about transition to more advanced languages, we might not use Java/C++ style syntax. Java makes a nice starting point for a sketching language because it's far more forgiving than C++ and also allows users to export sketches for distribution via the Web.

Programming languages used by artists and designers

The following list is a small sampling of the many languages in use by artists and designers. It is not possible to compile a comprehensive list, and this list does not aspire to meet that challenge. In addition to the languages mentioned here, there have been many historically important languages for the arts including GRASS, BEFLIX, Logo, AutoLISP, and PostScript. There are also many domain-specific languages that are not mentioned here.

ActionScript. ActionScript is a language written for Adobe's Flash software. Flash was originally created as Web animation software, and ActionScript was later added to provide scripting capabilities for MovieClips, the basic content unit of a Flash project. ActionScript is based on ECMAScript (the foundation of JavaScript), so knowledge of one transfers easily to the other. ActionScript is evolving rapidly and is becoming increasingly complex with each release. ActionScript 2.0 is based on ECMA-262 ECMAScript, which adds classes and strong data typing. Through the Flash Lite technology, ActionScript can be used to program content for mobile phones. *http://www.adobe.com/devnet/actionscript*

Arduino. (See Wiring) http://*www.arduino.cc*

BASIC. Originally designed in 1963 as a language to allow students in nontechnical fields to use computers, BASIC became prevalent on home computers in the 1980s. It was the first language learned by millions of children growing up with home computers at this time. BASIC was designed to be easy for beginners but also usable as a general-purpose language. There are many dialects of BASIC, including the PIC BASIC and PBASIC

languages for programming microcontrollers.
http://en.wikipedia.org/wiki/BASIC_programming_language

C, C++. The C language was developed in the early 1970s and is still used widely today. Many languages designed subsequently (including PHP and Processing) have used C as a model. In addition to its widespread use for writing system software and applications for PCs, it's a popular language for programming microcontrollers. C++ was designed to enhance the C language through the addition of object-oriented concepts. The ++ symbol in C means to add the number 1. The name C++ is geeky way to acknowledge its status as an enhanced version of C. Because C was so widely used, C++ became one of the most popular object-oriented languages. If well programmed, applications written in C and C++ are fast and efficient. Neither language has a built-in way of drawing; they are interfaced with a graphics library such as OpenGL to write images to the screen.
http://en.wikipedia.org/wiki/C_programming_language
http://en.wikipedia.org/wiki/C++_programming_language

ChucK. ChucK is an audio programming language for real-time synthesis, composition, and performance. Code can be added and modified while the program is running. The language offers precise timing control.
http://chuck.cs.princeton.edu

Design By Numbers (DBN). Design By Numbers was developed for teaching general programming concepts to artists and designers with no prior programming experience. DBN is an extremely minimal language and environment; thus, it is easy to learn but limited in its potential for creating advanced graphics applications. DBN was originated by John Maeda, director of the Aesthetics and Computation Group (ACG) at the MIT Media Laboratory. Processing also originated in the ACG because of Ben and Casey's involvement with the DBN project.
http://dbn.media.mit.edu

DrawBot. DrawBot, a language developed specifically for teaching, combines Python with a 2D graphics library and simple development environment. Images created can be output in different formats including PDF. This software is available only for Macintosh.
http://*www.drawbot.com*

Java. Java was created by Sun Microsystems in the 1990s as an alternative to C++. The language focuses on creating cross-platform programs with built-in support for networking. The popularity of Java dramatically increased as the Web emerged because of Java applets, programs that can run through a Web browser. In contrast to C and C++, Java programs are faster to write, but run more slowly. The Java language has grown at a tremendous rate since its conception and is now used for programming contexts including embedded devices, phones, server-side programs, and stand-alone applications.
http://java.sun.com

JavaScript. JavaScript was originally developed as a scripting language for enhancing Web pages. Despite its name, JavaScript is not directly related to the Java programming language. It was originally developed by Netscape and named LiveScript, but the name was changed to JavaScript around the time that Netscape began including Java with its Web browser. JavaScript was later sent to the ECMA standards body and codified as the ECMAScript standard. JavaScript is used as the scripting language for Scriptographer (p. 271) and Extend Script (Adobe's language for scripting its Illustrator, Photoshop, and InDesign software).
http://*www.mozilla.org/js*

Lingo. Lingo is the language written for Macromedia's Director software. Director was once the dominant environment for designers and artists making CD-ROM projects, but has declined in popularity during the Web era due to the success of Flash. It is still one of the environments most commonly used by artists and designers, and it has excellent libraries of code for extending its functionality. Lingo is integrated into a GUI environment that uses theater terms like "stage" and "cast" to describe different project elements. The Lingo language is characterized by its verbose English-like syntax. More recent features added to Director are accessible through JavaScript syntax. Director has been modified in recent years to support object-oriented structures and 3D graphics.
http://www.adobe.com/support/director/lingo.html

Max/MSP/Jitter. Max, named after the computer music pioneer Max Mathews, was originally a visual programming language for controlling MIDI data. The Max GUI is based on an analog synthesizer. Different modules (objects) are visually connected with patch cords to determine the data flow. The MSP objects were added ten years later to enable the software to generate live audio. The subsequent Jitter objects extended Max to control video and 3D graphics. Versions 4.5 and higher allow JavaScript and Java code to be used in tandem with the visual programming elements. Max/MSP/Jitter is commonly used for creating live audiovisual performances.
http://*www.cycling74.com/products/maxmsp, www.cycling74.com/products/jitter*

Maya Embedded Language (MEL). MEL is a scripting language used with Alias's Maya software. It is useful for automating repetitive tasks and for grouping sets of frequently used commands together into reusable scripts. The syntax is similar to C, and the language does not yet have object-oriented capabilities.
http://www.alias.com/maya, http://en.wikipedia.org/wiki/Maya_Embedded_Language

Mobile Processing. Mobile Processing is a variation of the Processing language for writing mobile phone programs. The graphics library is optimized to run on the simpler phone processors, and new functions are added to utilize unique elements of the phone such as multitap text input. The language is extended with libraries to interface with Bluetooth, SMS, and the phone's camera and audio capabilities.
http://mobile.processing.org

Perl. A goal of the Perl language is to make easy tasks easy and difficult tasks possible. It succeeds because it is a flexible and extensive language. The Perl syntax is a pastiche of many languages including C, awk, sed, sh, and others. Perl is used widely for Web development and network programming and is therefore sometimes called the "the duct tape of the Internet"; its popularity surged in the 1990s and later inspired Web scripting languages like PHP. Perl is excellent at parsing and manipulating text; its ability to process such data makes it useful for an art and design audience.
http://www.perl.org

PHP. PHP is a simple but powerful scripting language originally designed for programming dynamic Web content. The syntax is similar to C and is easily embedded within HTML. PHP is often used to read and write data from a database.
http://*www.php.net*

Pure Data (Pd). Pd is a visual programming language developed for creating computer music and live images. Pd was initiated by Miller Puckette, the father of Max, as an open-source alternative to the original proprietary software. It extends beyond the original Max with the additional of real-time audio synthesis. As in Max, programs are written with visual patches. As an open-source project, the Pd software and distributions contain many contributions from developers around the world. Pd is an extremely popular language for creating live audiovisual performances.
http://puredata.info

Python. Python is considered to be an excellent teaching language because of its clear syntax and structure. Python is typically used for nongraphic applications. It doesn't have a native graphics library, but graphics applications may be created by using graphics toolkits, some of which are cross-platform. The language is characterized by a small syntax and a huge variety of modules that extend the possibilities of the language. The DrawBot program and the TurboGears Web framework are both written with Python.
http://*www.python.org*

Quartz Composer. The Quartz Composer is a visual programming language included with Mac OS X for processing and rendering graphical data using OpenGL. The basic element of the language is a patch, the visual equivalent of a function. Input and output ports on a patch are connected with lines to define the flow of data within the program. Compositions, as programs written with Quartz Composer are called, can be run autonomously or incorporated into applications.
http://developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposer

Ruby. Ruby is an object-oriented scripting language. It has many features to process text files and to perform system management tasks. The Ruby syntax provides programmers great flexibility in structuring their code. This makes the language "expressive," but can also make it more difficult for other people to read. It has gained popularity because of Ruby on Rails, a framework for making Web applications. Ruby is a relatively new

language (it was created in 1995) and its user base is growing quickly.
http://*www.ruby-lang.org/en*

SQL (Structured Query Language). SQL is the most common programming language used to create, modify, retrieve, and manipulate database content. Its origins date back to 1969, but it only became standardized in 1986. While not a language for building applications per se, it provides a syntax for searching and collecting information from a database through queries and procedures.
http://en.wikipedia.org/wiki/Sql

SuperCollider. SuperCollider is an environment for real-time audio synthesis. It features a built-in programming language, an object-oriented class system, a GUI builder for creating a patch control panel, a graphical interface for creating wave tables and breakpoint envelopes, MIDI control, a large library of signal processing and synthesis functions, and a large library of functions for list processing musical data. The programming language has elements of the Smalltalk and C languages. This software is available only for Macintosh.
http://*www.audiosynth.com*

Wiring. Wiring is a language for programming microcontrollers. It's used to teach the basic concepts of working with electronics and skills in prototyping electronic devices. The language is based on Processing but tailored for electronics. Programs are developed within a modified version of the Processing Environment. When a Wiring program is compiled, it is translated into C code and then compiled as a C program. Wiring is also the programming language for the Arduino electronics boards.
http://*www.wiring.org.co*

vvvv. The vvvv website states, "vvvv is a multipurpose toolkit focusing on real-time video synthesis, connecting physical devices, and developing interactive media applications and systems."[3] It is a visual programming language with aspects similar to Max and Pure Data, but with a better designed interface and less emphasis on audio. It features hardware-accelerated 3D graphics and makes it easy to create multiscreen projections. This software is available only for Windows and is free for noncommercial use.
http://vvvv.meso.net

Notes

1. http://en.wikipedia.org/wiki/Alphabetical_list_of_programming_languages.
2. http://weblogs.media.mit.edu/llk/scratch/.
3. http://vvvv.org/tiki-index.php?page=executive+FAQ.

Appendix G

Code Comparison

The Processing programming language has similarities and differences with other commonly used languages. The table presented below compares elements of the Processing language to the Java, ActionScript, and Lingo languages. Lingo is the programming language for Macromedia's Director software. ActionScript is the language for Adobe's Flash software. These two languages are frequently used by designers and artists. Lingo was most popular in the mid-1990s and has gradually

Processing	*Java*
Color	
`background(0);`	`g.setColor(Color.black)` `fillRect(0, 0, size.width, size.height);`
`background(255);`	`g.setColor(Color.white)` `fillRect(0, 0, size.width, size.height);`
`background(255, 204, 0);`	`g.setColor(new Color(255, 204, 0));` `fillRect(0, 0, size.width, size.height);`
`stroke(255);`	`g.setColor(Color.white)`
`stroke(0);`	`g.setColor(Color.black)`
`stroke(255, 204, 0);`	`g.setColor(new Color(255, 204, 0));`
`fill(0, 102, 153);`	`g.setColor(new Color(0, 102, 153));`
Shape	
`point(30, 20);`	`g.drawLine(30, 20, 30, 20);`
`line(0, 20, 80, 20);`	`g.drawLine(30, 20, 80, 20);`
`rect(10, 20, 30, 30);`	`g.fillRect(10, 20, 30, 30);` `g.drawRect(10, 20, 30, 30);`

declined in the twenty-first century with the rise of ActionScript. Java is a general-purpose programming language in use within many domains. The Processing programming language is built on Java and therefore has many similarities to it. Additional comparisons to other languages are available by selecting the "Reference" option from the Help menu and clicking on the "Comparison" link.

ActionScript 2.0	*Lingo*
N/A	the stageColor = 255
N/A	the stageColor = 0
N/A	(the stage).bgcolor = rgb(255,204,0)
lineStyle(x, 0xFFFFFF, a, true, "none", "round", "miter", 1);	(the stage).image.draw(x1,y1,x2,y2, 0)
lineStyle(x, 0x000000, a, true, "none", "round", "miter", 1);	(the stage).image.draw(x1,y1,x2,y2, 255)
lineStyle(x, 0xFFCC00, a, true, "none", "round", "miter", 1);	(the stage).image.draw(x1,y1,x2,y2, rgb(255,204,0))
beginFill (0x006699);	(the stage).image.fill(left, top, right, bottom, rgb(0,102,153))
setPixel(30, 20, 0x000000)	(the stage).image.setPixel(30,20, rgb(0,120,153))
moveTo(x1,y1); lineTo(x2,y2);	(the stage).image.draw(0, 20, 80, 20, [#shapeType:#line])

moveTo(10,20);
lineTo(30,20);
lineTo(30,30);
lineTo(10,30);
lineTo(10,20);
	(the stage).image.fill(10,20,30,30, [#shapeType:#rect])
	(the stage).image.draw(10,20,30,30, [#shapeType:#rect])

Processing

Java

Data

```
int x = 70;   // Initialize
x = 30;  // Change value
```

```
int x = 70;   // Initialize
x = 30;  // Change value
```

```
float x = 70.0;
x = 30.0;
```

```
float x = 70.0f;
x = 30.0f;
```

```
int[] a = {5, 10, 11};
a[0] = 12;  // Reassign
```

```
int[] a = {5, 10, 11};
a[0] = 12;  // Reassign
```

Control

```
for (int a = 45; a <= 55; a++) {
  // Statements
}
```

```
for (int a = 45; a <= 55; a++) {
  // Statements
}
```

```
if (c == 1) {
  // Statements
}
```

```
if (c == 1) {
  // Statements
}
```

```
if (c != 1) {
  // Statements
}
```

```
if (c != 1) {
  // Statements
}
```

```
if (c < 1) {
  // Statements
}
```

```
if (c < 1) {
  // Statements
}
```

```
if (c >= 1) {
  // Statements
}
```

```
if (c >= 1) {
  // Statements
}
```

```
if ((c >= 1) && (c < 20)) {
  // Statements
}
```

```
if ((c >= 1) && (c < 20)) {
  // Statements
}
```

```
if (c >= 20) {
  // Statements 1
} else if (c == 0) {
  // Statements 2
} else {
  // Statements 3
}
```

```
if (c >= 20) {
  // Statements 1
} else if (c == 0) {
  // Statements 2
} else {
  // Statements 3
}
```

Structure

```
// Comment
```

```
// Comment
```

```
void doIt(int x) {
  // Statements
}

doIt(x);
```

```
public void doIt(int x) {
  // Statements
}

doIt(x);
```

ActionScript 2.0	*Lingo*
`var x:Number = 70;` `x = 30; // Change value`	`x = 70  -- Initialize` `x = 30  -- Change value`
`var x:Number = 70.0;` `x = 30.0; // Change value`	`x = 70.0  -- Initialize` `x = 30.0  -- Change value`
`var a:Array = [5, 10, 11];` `a[0] = 12; // Reassign`	`a = [5,10,11]` `a[1] = 12  -- Reassign`

`for (var a:Number = 45; a <= 55; a++) {`
`  // Statements`
`}`
	`repeat with a = 45 to 55`
	`  -- Statements`
	`end repeat`

`if (c == 1) {`
`  // Statements`
`}`
	`if c = 1 then`
	`  -- Statements`
	`end if`

`if (c != 1) {`
`  // Statements`
`}`
	`if not(c = 1) then`
	`  -- Statements`
	`end if`

`if (c < 1) {`
`  // Statements`
`}`
	`if c < 1 then`
	`  -- Statements`
	`end if`

`if (c >= 1) {`
`  // Statements`
`}`
	`if c >= 1 then`
	`  -- Statements`
	`end if`

`if ((c >= 1) && (c < 20)) {`
`  // Statements`
`}`
	`if c >= 1 and c < 20 then`
	`  -- Statements`
	`end if`

`if (c >= 20) {`
`  // Statements 1`
`} else if (c == 0) {`
`  // Statements 2`
`} else {`
`  // Statements 3`
`}`
	`if c >= 20 then`
	`  -- Statements 1`
	`else if c = 0 then`
	`  -- Statements 2`
	`else`
	`  -- Statements 3`
	`end if`

`// Comment`	`-- Comment`

`private function doIt (x:Number):Void {`
`  // Statements`
`}`
 	
`doIt(x);`
	`on doIt x`
	`  -- Statements`
	`end`
	
	`doIt x`

Processing

```
int square(int x) {
  return x*x;
}

square(X);
```

Java

```
public int square(int x) {
  return x*x;
}

square(X);
```

Input

```
mouseX
mouseY
```

```
/* Assuming there are two variables in
the program named mouseX and mouseY,
these values must be changed by the
programmer in the mouseMoved() and
mouseDragged methods. */

public void mouseMoved(MouseEvent e) {
  mouseX = e.getX();
  mouseY = e.getY();
}

public void mouseDragged(MouseEvent e) {
  mouseX = e.getX();
  mouseY = e.getY();
}
```

```
void mousePressed() {
  // Statements
}
```

```
public void mousePressed(MouseEvent e) {
  // Statements
}
```

```
if (key == 'a') {
  // Statements
}
```

```
public void keyPressed(KeyEvent e) {
  char key = e.getKeyChar();
  if (key == 'a') {
    // Statements
  }
}
```

```
void keyPressed() {
  // Statements
}
```

```
public void keyPressed(KeyEvent e) {
  // Statements
}
```

ActionScript 2.0	*Lingo*

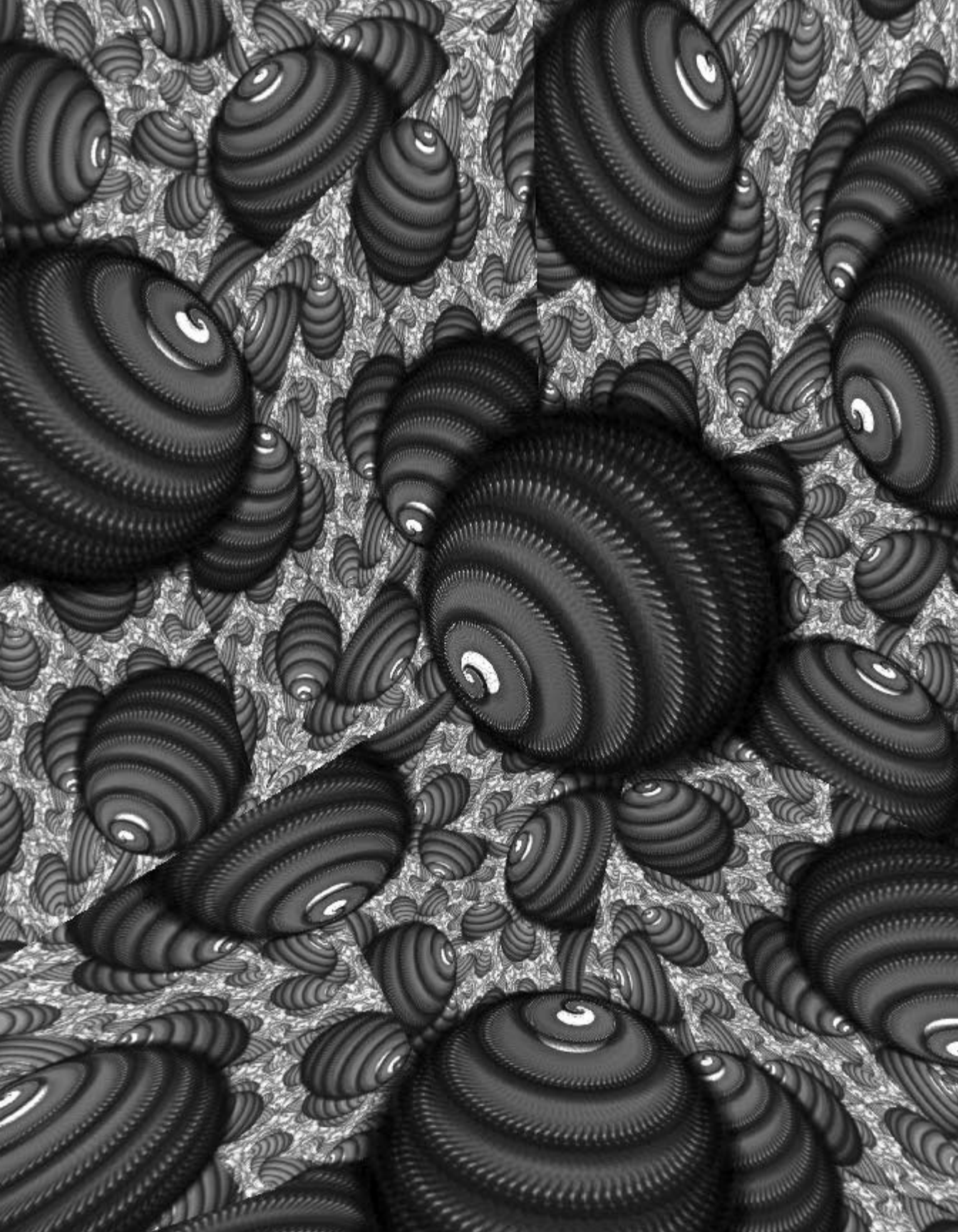
`function square(x:Number):Number {`
`  return x*x;`
`}`
 	
`square(x);`
	`on square x`
	`  return x*x`
	`end`
	
	`put square(x)`

`_xmouse` `_ymouse`	`the mouseH` `the mouseV`

`// Create a mouse listener object`
`var mouseListener:Object = new Object();`
`mouseListener.onMouseDown = function() {`
`  // Statements`
`};`
 	
`Mouse.addListener(mouseListener);`
	`on mouseDown`
	`  -- Statements`
	`end if`

`if ((chr(key.getAscii()) == 'a') {`
`  // Statements`
`}`
	`on keyDown`
	`  if the key = "a"`
	`  -- Statements`
	`  end if`
	`end`

`var myListener:Object = new Object();`
`myListener.onKeyDown = function () {`
`  // Statements`
`}`
`Key.addListener(myListener);`
	`on keyDown`
	`  -- Statements`
	`end`



Related Media

This book is an introduction to working with software in the domain of the visual arts. There are many related texts, websites, and software that explore these topics in greater depth. This list includes some that we feel are particularly important.

Color

Albers, Joseph. *The Interaction of Color*. Yale University Press, 1975.

Compelling book on color from the Bauhaus/Black Mountain/Yale artist-teacher.

Bourke, Paul. Personal website: Colour. http://astronomy.swin.edu.au/~pbourke/colour.

Diagrams, code, and explanations of many different color models.

Itten, Johannes. *The Elements of Color*. Van Nostrand Reinhold, 1970.

Trujillo, Drew. In The Mod: Color Analytics. 21 March 2006. http://www.inthemod.com.

Walch, Margaret, and Augustine Hope. *Living Colors: The Definitive Guide to Color Palettes through the Ages*. Chronicle Books, 1995. *Presents color palettes extracted from historical artworks and artifacts.*

Computer graphics (See page 545 for additional 3D references)

Ammeraal, Leendert. *Computer Graphics for Java Programmers*. John Wiley & Sons, 1998.

Hearn, Donald, and M. Pauline Baker. *Computer Graphics: C Version*. Second edition. Prentice Hall, 1986.

Foley, James D., and Andries van Dam, et al. *Computer Graphics: Principles and Practice in C*. Second edition. Addison-Wesley, 1995

OpenGL Architecture Review Board. *OpenGL Programming Guide*. Fourth edition. Addison-Wesley, 2003.

Original and definitive guide to OpenGL, but not for the beginning programmer. An earlier edition is available free online at http://www.opengl.org/documentation/red_book.

OpenGL Architecture Review Board. *OpenGL Reference Manual*. Fourth edition. Addison-Wesley, 2004.

An earlier edition is available free online at http://www.opengl.org/documentation/blue_book.

Computer vision (See page 561)

Drawing

Cohen, Harold. AARON. 21 March 2006. http://crca.ucsd.edu/~hcohen.

Links to Cohen's essays written about AARON, 1973-1999.

Klee, Paul. *Pedagogical Sketchbook*. Translated by Sibyl Moholy-Nagy. Frederick A. Praeger, 1953.

Whimsical journey through Klee's ideas about drawing.

Simon, John F. Jr. *Mobility Agents: A Computational Sketchbook*. Printed Matter and Whitney Museum of American Art, 2005.

CD of software applications exploring ideas about computational drawing.

Soda. MOOVL. 21 March 2006. www.moovl.com.

Drawn shapes are connected by springs and are affected by their environment.

Sutherland, Ivan. "Sketchpad: A Man-Machine Graphical Communication System." PhD dissertation, Massachusetts Institute of Technology, 1963.

Original documentation for the pioneering Sketchpad system. Available online at www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-574.pdf.

Robert Hodgin. *Video Spiral*, 2006. Live camera input is mapped onto 3D geometry. Image courtesy of the artist.

Electronics (See page 658)

Games

Crawford, Chris. *The Art of Computer Game Design*. McGraw Hill, 1983.

Online at www.vancouver.wsu.edu/fac/peabody/game-book/Coverpage.html and www.mindsim.com/MindSim/Corporate/artCGD.pdf.

Frasca, Gonzalo. Ludology.org. http://www.ludology.org.

Video game theory blog.

Gamasutra.com. The Art and Business of Making Games. http://www.gamasutra.com.

Salen, Katie, and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. MIT Press, 2004.

Salen, Katie, and Eric Zimmerman, eds. *The Game Design Reader: A Rules of Play Anthology*. MIT Press, 2005.

Sudnow, David. *Pilgrim in the Microworld*. Warner Books, 1983.

History of software as art

Burnham, Jack. *Great Western Salt Works: Essays on the Meaning of Post-Formalist Art*. George Braziller, 1974.

Davis, Douglas. *Art and the Future: A History/Prophecy of the Collaboration between Science, Technology, and Art*. Henry Holt & Company, 1975.

Digital Art Museum. 20 July 2006. http://www.dam.org.

Online resource for the history and practice of digital art.

Franke, H. W. *Computer Graphics Computer Art*. Phaidon, 1971.

Discusses methods for creating computer art and introduces the brief history preceding this early publication.

Glimcher, Marc. *Logical Conclusions: 40 Years of Rule-Based Art*. Pace Wildenstein, 2005.

Lippard, Lucy R. *Six Years: The Dematerialization of the Art Object, 1966 to 1972*. University of California Press, 1973.

Paul, Christiane. *Digital Art*. Thames & Hudson, 2003.

Well-structured overview of the field.

Medien Kunst Netz. 10 July 2006. http://www.medienkunstnetz.de.

Online repository of historic and contemporary media art concepts and works.

Reichardt, Jasia. *The Computer in Art*. Studio Vista, 1971.

Small book (98 pages) introducing the relation between software and image.

Reichardt, Jasia. *Cybernetics, Art, and Ideas*. New York Graphic Society, 1971.

UbuWeb. 20 July 2006. http://www.ubu.com.

Online "resource dedicated to all strains of the avant-garde, ethnopoetics, and outsider arts."

Whitney, John. *Digital Harmony: On the Complementary of Music and Visual Art*. Byte Books (McGraw-Hill), 1980.

Wilson, Mark. *Drawing with Computers*. Putnam, 1985.

Surveys the technology of the era and presents many examples for drawing to plotters and screen.

Youngblood, Gene. *Expanded Cinema*. Dutton, 1970.

Documents the state of experimental film and animation circa 1970. Part 4 introduces "Cybernetic Cinema and Computer Films."

Image

Efford, Nick. *Digital Image Processing: A Practical Introduction Using Java*. Addison-Wesley, 2000.

Excellent introduction to the concepts, math, and code of image processing.

Myler, Harley R. *The Pocket Handbook of Image Processing Algorithms*. Prentice Hall, 1993.

Small black book of essential image processing algorithms.

Sontag, Susan. *On Photography*. Anchor Books, 1977.
Thoughtful and provocative essays about photographic images and their role in culture.

Information visualization

Bertin, Jacques. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, 1983.
English translation and later edition of French text first published in 1967. A seminal work in the field.
Card, Stuart K., et al., eds. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.
Compiled technical papers on information visualization.
Fry, Benjamin. "Organic Information Design." Master's thesis, Massachusetts Institute of Technology, Program in Media Arts & Sciences, 2000.
Fry, Benjamin. *Computational Information Design*. PhD dissertation, Massachusetts Institute of Technology, Program in Media Arts & Sciences, 2004.
Tufte, Edward. *Envisioning Information*. Graphics Press, 1990.
Tufte, Edward. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

Input

Apple Computer Inc. *Macintosh Human Interface Guidelines*. Addison-Wesley, 1992.
Introduces the design considerations and elements of designing human-computer interfaces.
Related content online at http://developer.apple.com/referencelibrary/UserExperience.
Engelbart, Douglas, and Bill Paxton. NLS Demo. Joint Computer Conference, San Francisco Convention Center, 9 December 1968.
Video documentation of the seminal demonstration introducing the mouse input device. Online at http://sloan.stanford.edu/MouseSite/1968Demo.html.
Maeda, John. *Reactive Books*. Digitalogue, 1994–1999. *Online at www.maedastudio.com/2004/rbooks2k.*
Software exploring sound, mouse, clock, keyboard, and video input channels.
Stephenson, Neal. *In the Beginning Was the Command Line*. Harper Perennial, 1999.
Online at http://www.cryptonomicon.com/beginning.html.

Math

Bourke, Paul. Personal website: Curves. http://astronomy.swin.edu.au/~pbourke/curves.
Online repository of curve equations including images and explanation.
Famous Curves Index. www-history.mcs.st-and.ac.uk/Curves/Curves.html.
Collection of equations and corresponding software to control the shape of curves.
Lial, Margaret L., E. John Hornsby, Jr., and David I. Schneider. *College Algebra*. Seventh edition. Addison-Wesley, 1997.
Tan, Manny, et al. *Flash Math Creativity*. Friends of Ed, 2002.
Collection of math techniques for producing images and motion.
Van Lerth, James, and Lars Bishop. *Essential Mathematics for Games and Interactive Applications*. Morgan Kaufmann, 2004.
Weisstein, Eric. MathWorld. http://mathworld.wolfram.com.
Extensive online math resource. Includes images, equations, and applets.

Mobile computing (See page 631)

Motion

Peters, Keith. *ActionScript Animation: Making Things Move!* Friends of Ed, 2005.

Full of great algorithms for programming motion.

Muybridge, Eadweard. *Animals in Motion*. Dover, 1957.

Sequential photographs revealing details of motion.

Laybourne, Kit. *The Animation Book: A Complete Guide to Animated Filmmaking; From Flip-Books to Sound Cartoons to 3-D Animation*. Revised edition. Three Rivers Press, 1998.

General introduction to techniques of animation across many media.

Lieberman, Zachary. Making Things Move. Workshop at Medialab Madrid. 20–22 June 2005. www.thesystemis.com/makingThingsMove.

Code examples from a motion programming workshop.

Russett, Robert, and Cecile Starr. *Experimental Animation: Origins of a New Art*. Da Capo Press, 1976.

Excellent text and visual documentation of pioneers of experimental animation.

Thomas, Frank, and Ollie Johnston. *Disney Animation, The Illusion of Life*. Abbeville Press, 1981.

In-depth introduction to principles of character animation.

Network (See page 576)

Processing.org

Processing.org. http://www.processing.org.

Official Processing website including an exhibition, reference, examples, and software downloads.

Processing Hacks. http://www.processinghacks.com.

Documenting advanced Processing tricks and hacks. Led by Tom Carden and Karsten Schmidt (a k a toxi).

Processing Blogs. www.processingblogs.org.

Blog aggregator site curated by Tom Carden.

Processing.org del.icio.us tag. http://del.icio.us/tag/Processing.org.

Shape

Dondis, Donis A. *A Primer of Visual Literacy*. MIT Press, 1973.

Comprehensive introduction to basic elements and techniques of visual messages.

Hofmann, Armin. *Graphic Design Manual: Principles and Practice*. Van Nostrand Reinhhold, 1965.

Elegant book from a master teacher-designer.

Itten, Johannes. *Design and Form: The Basic Course at the Bauhaus and Later*. Revised edition. John Wiley & Sons, 1975.

Moholy-Nagy, Laszlo. *Vision in Motion*. Paul Theobald, 1947.

Simulation

Boden, Margaret A., ed. *The Philosophy of Artificial Life*. Oxford University Press, 1996.

Excellent collection of noteworthy essays.

Braitenberg, Valentino. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, 1984.

Playful text about imaginary vehicles and their relation to biology.

Flake, Gary William. *The Computational Beauty of Nature*. MIT Press, 1998.

Gardner, Martin. "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life.'" Scientific American 223 (October 1970): 120–123.

Levy, Steven. *Artificial Life: The Quest for a New Creation*. Pantheon Books, 1992.
Friendly introduction to AL, with vivid profiles of its founders.
Kelly, Kevin. *Out of Control: The New Biology of Machines, Social Systems, and the Economic World*. Addison-Wesley, 1994.
Resnick, Mitchel. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1997.
Documents ideas behind the StarLogo language for teaching children about decentralized systems.
Whitelaw, Mitchell. *Metacreation: Art and Artificial Life*. MIT Press, 2004.
Sims, Karl. "Evolving Virtual Creatures." *Computer Graphics*. Proceedings of Siggraph '94, July 1994, pp. 15–22.
Brief technical explanation of an important AL work.
Wolfram, Steven. *A New Kind of Science*. Wolfram Media, 2002.

Software data, control, structure

Downey, Allen B. *How to Think Like a Computer Scientist*. http://ibiblio.org/obp/thinkCS/java.php.
Taylor, David A. *Object Technology: A Manager's Guide*. Second edition. Addison-Wesley, 1998.
Introduces object-oriented programming as a concept and defines its attributes and advantages.
Flanagan, David. *Java in a Nutshell*. Fifth edition. O'Reilly Media, 2005.
Flanagan, David. *JavaScript: The Definitive Guide*. Fourth edition. O'Reilly Media, 2001.
Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
Oualline, Steve. *Practical C++ Programming*. Second edition. O'Reilly Media, 2003.
Kernighan, Brian, and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.
Includes techniques and advice for writing better software.
Maeda, John. *Design By Numbers*. MIT Press, 2001.
A fresh, clear introduction to core concepts of computer graphics and software.
Kernighan, Brian, and Dennis Ritchie. *C Programming Language*. Second edition. Prentice Hall, 1998.
Dense, canonical introduction to the C language.
Prata, Stephen. *C Primer Plus*. Fifth edition. Sams, 2004.
Practical, well-written introduction to the C language.
Sun Microsystems. *The Java Tutorial: Learning the Java Language*. http://java.sun.com/docs/books/tutorial/java.

Sound (See page 599)

Software culture & theory

Fuller, Matthew. *Behind the Blip*. Autonomedia, 2003.
Galloway, Alexander R. *Protocol: How Control Exists after Decentralization*. MIT Press, 2004.
Gere, Charlie. *Digital Culture*. Reaktion Books, 2002.
McCullough, Malcolm. *Abstracting Craft: The Practiced Digital Hand*. MIT Press, 1997.
Thoughtful reflection on the nature of craft and production utilizing software.
Maeda, John. *Maeda@Media*. Thames & Hudson, 2000.
Maeda, John. *Creative Code: Aesthetics + Computation*. Thames & Hudson, 2004.
Visual introduction to the work of the MIT Media Lab's Aesthetics and Computation Group.
Manovich, Lev. *The Language of New Media*. MIT Press, 2001.
Ludovico, Alessandro. *Neural* magazine and website. www.neural.it/english.

Magazine and website featuring reviews and articles on the topics of new media art, electronic music, and hacktivism.

Packer, Randall, and Ken Jordan, eds. *Multimedia from Wagner to Virtual Reality*. W.W. Norton, 2001.

Ploug, Kristine, and Thomas Petersen, eds. Artificial.dk. http://www.artificial.dk.

Online interviews, articles, and reviews of software art, net art, and other forms of computer arts.

Alexander, Amy, Olga Goriunove, Alex McLean, and Alexei Shulgin. Runme.org. http://www.runme.org.

Eclectic online software art repository.

Søndegaard, Morton. *Get Real: Art + Real-time: History, Practice, Theory*. George Braziller, 2005.

Wardrip-Fruin, Noah, and Nick Montfort. *The New Media Reader*. MIT Press, 2003.

Extensive introduction to the origins and current practices of new media through essays, software, and video.

Watz, Marius, ed. Generator.x: Software and Generative Strategies in Art and Design. http://www.generatorx.no.

Conference, exhibition, and Web log dedicated to exploring and documenting generative strategies.

Typography

Blockland, Erik van, and Just van Rossum. *LettError*. Drukkerij Rosbeek, 2000.

Bringhurst, Robert. *The Elements of Typographic Style*. Version 3.0. Hartley & Marks, 2004.

Impressive compendium of typographic history, convensions, and technologies.

Kunz, Willi. *Typography: Macro- and Micro Aesthetics*. Niggli, 1998.

Lupton, Ellen. *Thinking with Type: A Critical Guide for Designers, Writers, Editors, & Students*. Princeton Architectural Press, 2004.

Compact, clear introduction to typography for designers, writers, editors, and students. Information online at www.thinkingwithtype.com.

Ruder, Emil. *Typography*. Niggli, 1967.

Thorough introduction to the formal space of typographic exploration.

Small, David. "Navigating Large Bodies of Text." IBM Systems Journal 35, no. 3–4 (1996).

Documents Small's influential Shakespeare project and other projects of the MIT Media Lab VLW. Available online at www.research.ibm.com/journal/sj/353/sectiond/small.pdf.

Weingart, Wolfgang. *My Way to Typography*. Lars Müller Publishers, 2000.

Generously produced publication exposing the thoughts of a master of typography and form.

Glossary

This list defines programming terminology and redefines common words that are used differently within the context of software.

abstraction Refers to hiding details of a process to focus on the result. For example, the `line()` function abstracts the many lines of code necessary to draw a line to the screen, so the programmer can focus on the line's position.

additive color Color system for working with light. The additive primary colors red, green, and blue are combined to produce millions of variations. The absence of light is black, and adding the primaries together creates white.

AIFF (Audio Interchange File Format) Audio file format developed by Apple. Stores uncompressed data in pulse-code modulation (PCM) format.

algorithm A series of instructions that perform a mathematical task. Usually used to refer to a complicated bit of code that solves a problem like sorting a list or searching for text.

alpha The opacity component of a color value. By default, the value 255 makes a color entirely opaque and 0 sets a color as entirely transparent.

antialiasing Minimizing the aliasing (jagged edges) within a low-resolution display to simulate a higher-resolution image.

API (application programming interface) A set of functions that comprise the way to use a programming system. The Processing API consists of commands like `line()` and `point()`, and is referred to as the Processing Language.

applet Java program that can run within a compatible Web browser. Processing programs can be exported as applets.

array A list of data elements referenced with one name. Each element is accessed according to its order within the list.

ASCII (American Standard Code for Information Interchange) Code for associating the letters, numbers, and punctuation in the English language with numeric values. For example, *K* is 75 and *Y* is 89.

bit The most basic information unit in computing. Often represented as a 0 or 1.

block A group of code defined by matching braces, the `{` and `}` characters. Blocks are used to group code into classes, functions, `if` structures, and `for` structures.

bug An error within a program that causes a program to not run or to behave differently than intended by the programmer.

byte A byte is a group of 8 bits.

class A template defining a related group of fields and methods. Classes are the building blocks of object-oriented programming. An object is an instance of a class.

color depth Number of bits used to define a color. An 8-bit number can be values between 0 and 255 (2^8). A 4-bit number can be values between 0 and 15 (2^4).

compile To translate source code into executable software. When a Processing program is run, it is translated from code notation into a notation that can be run by a computer. This is called compilation.

data type The category of data that can be stored within a variable. For example, individual letters can be stored in variable of the `char` data type and whole numbers can be stored in variables of the `int` data type.

debug To remove errors from a program.

delimit To separate elements of data within a file. For example, a tab-delimited file separates data with the tab character.

dot operator The period character (.). Fields and methods of a class are accessed with the dot operator.

dpi (dots per inch) A measurement of printing resolution. A higher DPI printer can produce clearer images of higher resolution.

encapsulation Technique of hiding the data and the functions that operate on the data within a class. A class can be thought of as a black box, where the implementation within the class is less of a focus than how the class is used. The internal code of a class can be changed, while the way it interacts with other elements of the program can remain unchanged. Related to *abstraction*.

escape sequence A means of specifying unprintable characters (such as Tab or Enter) or quotes inside a String constant (which is defined using quote characters). Inside text, the combination of the `\` (backslash) character with another character. The backslash begins the escape sequence and the second character defines the meaning. For example, the sequence `\t` is a tab.

event An action such as a key being pressed, the mouse moving, or a new piece of data becoming available to be read. An event interrupts the normal flow of a program to run the code within an event block.

expression A combination of operators, variables, and literals. An expression always has a value, determined by its elements. For example, the expression `6/2` evaluates to `3` and the expression `5>4` evaluates to `true`. An expression can be as simple as a single number or can contain a long sequence of elements.

field A variable defined within a class.

file A collection of data referenced as a unit.

file format A specific way to store data within a computer file. There are different formats for storing images, sound, text, etc.

function A modular program unit within a large program. Functions have parameters to define their actions and can return values. In other programming languages functions may be called subroutines or procedures. A *method* is a function that belongs to a class.

function prototype Defines the parameters (inputs) for a function and their data types.

GIF (Graphics Interchange Format) Image file format commonly used for displaying graphics on the Web. The format supports compression, multiple color depths, and 1-bit transparency.

GUI (graphical user interface) Computer interface in which users control the machine by manipulating visual icons.

hex Abbreviation for hexadecimal.

hexadecimal Base-16 number system utilizing the symbols `0–9` and `A–F`. In Processing, a hexadecimal value is prefaced with a `#` or `0x`. Hexadecimal notation is often used to define color values. For example, the RGB color value (`0, 102, 153`) is converted to hexadecimal notation as `#006699`.

HSB (hue, saturation, brightness) Color model that defines a value through the quantities of hue (color), saturation (intensity), and brightness (light or dark). A more intuitive model than RGB.

IDE (integrated development environment) Software that assists people in the activity of programming. An IDE usually has a text editor, a compiler and/or interpreter, and a debugger. The Processing IDE is called the Processing Development Environment (PDE) or the Processing Environment.

inheritance A property of the relationship between classes and their sub- and superclasses. A class automatically contains the fields and methods defined in its superclass. If a class is a subclass of another, it inherits these components of that class.

instance An object of a particular class. For example, in code 43-03 (p. 400), the *sp* object is an instance of the `Spot` class. An instance is created with the `new` keyword.

instance variable A variable created when an object is instantiated from a class. Each object has its own instance of each variable defined in the class template.

JAR (Java ARchive) File format for grouping Java classes into a single unit. These files can be opened with any program that can open a ZIP file. Processing sketches are exported as JAR files.

JPEG (Joint Photographic Experts Group) Image format that compresses photographic images well. Common format for display on the World Wide Web. These files use the .jpg extension.

keyword A word used by a programming language. This word cannot be used for names of variables or functions.

memory A computer component that stores data. RAM (random access memory) is fast, but data is only stored there temporarily while a computer is on. Memory is stored more permanently on hard disk drives (HDD) that save data on magnetic disks. Small devices such as mobile phones, digital audio players, and USB drives use Flash memory, a technology that electronically erases and reprograms data.

method A function defined within a class.

new Used to create a new instance of an object. For example: `sp = new Spot()` (p. 400)

null Specifies an undefined value. An object variable that has not been assigned contains null. Null can also be assigned to a variable to set it empty.

object An instance of a class. All objects created from one class have the same field names and methods, but the variables of each can contain different data.

operand Data that is operated on by an operator. The operands in the expression `4 + 5` are the numbers 4 and 5.

operator A symbol that performs an operation. The *, =, +, %, >, and ! Symbols are all operators. Processing has arithmetic, relational, logical, and bitwise operators.

packet A block of formatted information transferred over a computer network. A packet has a header, data, and a trailer. Large pieces of data are broken down into multiple packets, each sent over the network separately and reassembled when they arrive at their destination.

parameter Data input to a function that affects the output. For example, the `point()` function can have two or three parameters to set the *x*, *y*, and *z* coordinates. The prototype for a function shows the number and data types of the parameters for a function.

PDE (Processing Development Environment) The Processing application, including the text editor, menu, toolbar, message area, and tabs. Also the name of the Processing file format (*.pde*).

pixel One color element (picture element) on a computer monitor or of a digital image.

PNG (Portable Network Graphics) Highly flexible image file format capable of variable levels of transparency. Developed as a successor to GIF.

PPI (pixels per inch) The pixel density of a computer screen.

radian Angle measurement in relation to π. The measurement of π is equivalent to 180 degrees, and 2π is equivalent to 360 degrees. The π symbol is represented within Processing as the constant `PI`.

relational expression An expression comprised of a relational operator and values to its left and right. A relational expression always evaluates to true or false. For example, the expression `4 < 5` evaluates to true and `4 > 5` evaluates to false.

relational operator An operator such as > (greater than), < (less than), and `!=` (not equal to) that determines the relation between the values to the left and right of the symbol.

return Used within a function to note the value that is returned as its result. The return statement is typically the last line of code in a function.

RGB (red, green, blue) Color model that defines a spectrum of colors in terms of their red, green, and blue components. RGB is the default color system used in Processing. It is considered less intuitive than HSB color because it's based on technical, rather than perceptual, attributes.

scope The region of a program where a variable can be accessed. A variable can be accessed within the block where it is defined and in all blocks defined within its block.

sketch Refers to a program written with Processing. Because Processing programs are intended to be written quickly and casually, they are often referred to as software sketches.

stack A data structure that stores elements in order so that they can be added and removed only from the top. Data is pushed (saved to the stack) and popped (removed from the top of the stack). The Processing functions `pushMatrix()` and `popMatrix()` perform this operation on the stack that controls the position, scaling, and rotation of the coordinate system.

statement A complete instruction to the computer. Statements are the primary building blocks of a program. A statement can define a variable, assign a variable, run a function, or construct an object. A statement always has a semicolon at the end.

statement terminator The semicolon symbol. Marks the end of a statement.

subclass A class derived from another (its superclass). A subclass inherits the template of its superclass.

super A keyword used within a subclass to refer to its superclass.

superclass A class that another is derived from.

TARGA Image file format that can store images without compression and with varying levels of transparency. These files use the *.tga* extension.

this A reference to the current object; also used within an object to refer to itself. For example, if a variable *x* is referred to within its own object, the code *`this.x`* can be used.

TIFF (Tagged Image File Format) Image file format used to store high-quality images without compression. These files use the *.tif* extension.

variable A data element that is referenced with a name. Every variable has a value, data type, and scope.

vertex A point that terminates, lies at the intersection of, or defines the shape of a line or curve.

VLW The Processing font format. The VLW (Visual Language Workshop) was a research group at the MIT Media Laboratory from 1985 to 1996. The researchers created the font format to display anti-aliased typography in 3D. Because of its usefulness in interactive graphics, it was made part of Processing, and named VLW in reference to its origin.

void Used in a function declaration to state that the function does not return a value (does not output data).

WAV Audio file format developed by Microsoft and IBM. Stores uncompressed data in pulse-code modulation (PCM) format.

XML (eXtensible Markup Language) Data formatting standard that is easy to read and customize.

Code Index

This index contains all of the Processing language elements introduced within this book. The page numbers refer to the first use.

! (logical NOT), 57
!= (inequality), 52
% (modulo), 45
&& (logical AND), 57
() (parentheses)
  for functions, 18
  for precedence, 47
* (multiply), 44
*= (multiply assign), 49
+ (addition), 43
++ (increment), 48
+= (add assign), 48
, (comma), 18
- (minus), 44
-- (decrement), 48
-= (subtract assign), 48
. (dot), 107
/ (divide), 44
/= (divide assign), 49
/* */ (comment), 18
// (comment), 17
; (semicolon), 19
< (less than), 51
<= (less than or
    equal to), 52
= (assign), 38
== (equality), 52
  for String objects, 109
> (greater than), 51
>= (greater than
    or equal to), 52
[] (array access), 301
  2D arrays, 312
  arrays of objects, 406
{} (braces), 53
  and variable scope, 178
|| (logical OR), 57
(hex color), 93

abs(), 241
alpha(), 338
ambient(), 533
ambientLight(), 533
append(), 309
arc(), 124
arraycopy, 310
Array, 301
  length, 304
atan2(), 243

background(), 31
beginRaw(), 531
beginRecord(), 607
beginShape(), 69
bezier(), 30
bezierVertex(), 75
blend(), 351
blendColor(), 352
blue(), 337
boolean, 38
boolean(), 106
brightness(), 338
byte, 38
byte(), 106

camera(), 531
Capture, 556
ceil(), 49
char, 38, 102
char(), 106
class, 395
Client, 567
color, 38, 89
color(), 89
colorMode(), 91
constrain(), 237
copy(), 353
cos(), 118
createGraphics(), 614
createImage(), 362
createWriter(), 423
cursor(), 213
curveVertex(), 74

day(), 249
degrees(), 117
directionalLight(), 536
dist(), 238
draw(), 173

ellipse(), 30
ellipseMode(), 34
else, 55
else if, 56
endRaw(), 531
endRecord(), 607
endShape(), 69
exit(), 422
expand(), 309
extends, 456

false, 38
fill(), 32
filter(), 347
float, 37
float(), 106
floor(), 49
for, 61
frameCount, 173
frameRate(), 173

get(), 321
green(), 337

HALF_PI, 117
height, 40
hour(), 245
HSB, 89
hue(), 338

if, 53
image(), 96
int, 37
int(), 107
key, 225
keyCode, 227
keyPressed, 224
keyPressed(), 232
keyReleased(), 232

lerp(), 81
lightSpecular(), 536
line(), 27
loadFont(), 112
loadImage(), 96
loadPixels(), 356
loadStrings(), 428
loop(), 235

map(), 81
mask(), 354
max(), 50
millis(), 248
min(), 50
minute(), 245
month(), 249
mouseButton, 212
mouseDragged(), 229
mouseMoved(), 229
mousePressed, 212
mousePressed(), 229
mouseReleased(), 229
mouseX, 205
mouseY, 205

new
  for arrays, 303
  for objects, 399
nf(), 422
noCursor(), 213
noFill(), 33
noise(), 130
noiseSeed(), 131
noLoop(), 178
norm(), 80
noSmooth(), 33
noStroke(), 33
noTint(), 97

Object, 107, 395

PFont, 112
PI, 117
PImage, 96
pixels[], 356
pmouseX, 208
pmouseY, 208
point(), 25
pointLight(), 536
popMatrix(), 134
pow(), 80
print(), 20
println(), 20
PrintWriter, 423
  close(), 423
  flush(), 423
  println(), 424
pushMatrix(), 134

quad(), 29
QUARTER_PI, 117

radians(), 117
random(), 127
randomSeed(), 129
rect(), 29
rectMode(), 34
red(), 337
redraw(), 235
return, 194
RGB, 89
rotate(), 137
round(), 50

saturation(), 338
save(), 368
saveFrame(), 369
saveStrings(), 422
scale(), 138
second(), 245
Server, 567
set(), 324
setup(), 177
shorten(), 309
sin(), 118
size(), 24
  with P3D, 528
  with OPENGL, 528
  with PDF, 607
smooth(), 33
specular(), 536
split(), 429
splitTokens(), 430
spotLight(), 536
sq(), 79
sqrt(), 79
str(), 107
String, 103
  length(), 108
  endsWith(), 108
  equals(), 109
  startsWith(), 108
  substring(), 109
  toCharArray(), 108
  toLowerCase(), 109
  toUpperCase(), 109
stroke(), 32
strokeCap(), 33
strokeJoin(), 33
strokeWeight(), 33
super, 456

text(), 112
textAlign(), 115
textFont(), 112
textLeading(), 115
textSize(), 114
texture(), 536
textWidth(), 116
tint(), 97
translate(), 133
triangle(), 27
true, 38
TWO_PI, 117

updatePixels(), 356

vertex(), 69
void, 187

width, 40

year(), 249

Index

This index contains mostly people, software, artwork, and programming languages. For topics, see the table of contents (pp. vii–xvii); for code, see the Code Index.

1:1 (Jevbratt), 566
3M Corporation, 553
3 Stoppages Étalon (Duchamp), 127
7–11 Email list, 563

AARON, 218
Aesthetics and Computation Group (ACG), xxiii, 682
Achituv, Romy, 549
ActionScript, 158, 166, 522–523, 565, 680–681, 686–687, 689, 691
Adair, Sandra, 384
Adobe, 4, 169, 683
Adobe After Effects, 166, 327, 379, 387
Adobe Flash, 157–158, 165–166, 267–268, 275, 278, 327, 436, 564–565, 624, 629, 642, 680–681, 683, 686, 701
Adobe Flash Lite, 624, 681
Adobe Garamond (font), 112
Adobe Illustrator, xxiii, 30, 77, 166, 143, 217, 271, 273, 607–608, 683
Adobe Photoshop, xxiii, 95, 166, 268, 276, 347, 355, 360, 384, 387–388, 391–392, 607–608, 611, 683
Adobe Premiere, 391–392
Adobe Streamline, 166
AAC (Advanced Audio Coding), 585
AIFF (Audio Interchange File Format), 585–586, 699
Aldus PageMaker, 605
Alexander, Ryan, 380
Alias Maya, 379, 387–388, 537, 680
AltSys, 170
Andrade, Laura Hernandez, 4
Apple IIe, xxiii
Apple Audio Units (AU), 591
Apple Computer, 3, 111, 537, 585, 699
Apple Logic Audio, 503, 591
Apple Mac G3, 383
Apple Mac G4, 383
Apple Macintosh (Mac), 9–11, 95, 111–112, 169, 205, 227, 367, 383, 521, 568–569, 574, 604, 639, 665, 682, 685
Apple Mac Mini, 639
Apple Mac OS, 264, 435, 665–666, 681
Apple Mac OS X, 16, 170, 435, 645, 649, 684
Apple QuickTime, 367, 383–384, 387–388
AppleScript, 681
Arduino, 521, 633, 640, 641, 645–646, 648–649, 681, 685
Arp, Jean, 127
Ars Electronica Festival, 618
ART+COM, 498
ASCII (American Standard Code for Information Interchange), 102–103, 226–227, 549, 565, 664–668, 670, 691, 699
Athena, 387
ATI, 537
AT&T/Bell , 564
Audacity, 591
AutoCAD, 217, 529, 537
Autodesk 3ds Max, 268, 276, 391–392, 537
AutoDesk Revit, 537
AutoLISP, 522, 681
Autonomedia, 564
Avid/Digidesign Pro Tools, 591
AVR (Atmel), 640
awk, 517, 684

Babbitt, Milton, 580–581
Bach, J. S., 581
Bailey, Chris, 581
Balkin, Amy, 267
Baran, Paul, 564
Barr, Alfred, 291
Barragán, Hernando, 633
BASIC, xxiii, xxiv, 152, 264, 522, 604–605, 640, 642, 681
BASIC Stamp 2 (Parallax), 640
BasicX–24 (NetMedia), 642
Bass, Saul, 327
Baumgärtel, Tilman, 564
Bauhaus, 149
BBC Acorn Archimedes, 264
Beach Culture, 605
Beethoven, Ludwig van, 581
BEFLIX, 315, 681
Bell Laboratories, 315, 580–581, 604
Bentley Systems GenerativeComponents, 537
Berliner, Emile, 579
Berlow, David, 170
Bernard (a k a Flip 1), 508
BIAS Peak, 591
BigEye, 554
Binary Runtime Environment for Wireless (BREW), 625
Binary space partition (BSP), 527
Binder, Maurice, 327
bitforms gallery, 164, 166–167, 525, 547, 603, 633
Bittorent, 571
Blackwell, Lewis, 605
Blender, 276, 576
Blinkenlights (Chaos Computer Club), 618
Blonk, Jaap, 511
Bluetooth, 619, 621–622, 624, 641, 645, 683
Blyth, Steven, 512
Boids (Reynolds), 295, 473, 475, 497
Boole, George, 38, 61, 669
Boolean algebra, 38
Boulez, Pierre, 581
Braitenberg, Valentino, 473–474
Brakhage, Stan, 413
Brecht, Bertolt, 564
Brooklyn Academy of Music (BAM), 515–516
Brown, Robert, 295
Brownian motion, 295
Brunelleschi, Filippo, 525
Bunting, Heath, 563–564
Bureau of Inverse Technology, 548, 634
Burke, Phil, 592
Burton, Ed, 263–264, 413, 499
Byrne, David, 581

C, 7, 264, 515–517, 522–523, 592, 640, 642, 682–685, 693, 697
C++, 264, 271, 383, 507–508, 511–512, 515–516, 522–523, 555, 592, 599, 640, 679, 681–682
CAD (computer–aided drawing software), 217, 526, 537–538
Cage, John, 127, 579
CalArts School of Art, 564
California Institute of Technology (Caltech), 388, 549
Cameron, Dan, 387
Campbell, Jim, 549
Carmack, John ,525
Carnegie Mellon University, xxi
Carnivore, 566, 568–569
Carson, David, 605
Cascading Style Sheets (CSS), 93
CCRMA Synthesis ToolKit (STK), 592
Chang, Zai, 6
Cheese (Möller), 549
Cho, Peter, 257, 327
CIA World Fact Book, 267
Citron, Jack, 315
CityPoems, 617, 624
ChucK, 592, 682
Cloaca (Delvoye), 461
Clash of the Titans, 387
Close, Chuck, 606
CODE (Petzold), 648
Cohen, Harold, 218
Columbia–Princeton Electronic Music Center, 580
Commodore C–64, 272
Commodore VC–20, 272
Common Lisp, 592
Complexification.net, 6, 157
Computational Beauty of Nature, The (Flake), 469
Computers and Automation, 603
Computer Clubhouse, 680
Computer Lib / Dream Machines (Nelson), 3
Computer Vision Homepage (Huber), 552
Coniglio, Mark, 512
"Constituents for a Theory of the Media" (Enzensberger), 564
Conway, John, 461, 463, 467–468, 475
Cook, Perry, 592
Cooper, Muriel , 327
Cope, David, 581
CorelDRAW, 608
Cosic, Vic 563–564
Costabile, Sue (SUE.C), 503–504
Craighead, Alison, 618
Crawford, David, 316
Crystal Castle, 525
Csikszentmihályi, Chris, 507–508, 634
CSIRAC, 580
Csuri, Charles, 217
Cuba, Larry, 1, 315
Cullen, Mathew, 379–380
CV.Jit, 554
Cybernetic Serendipity, 101, 603
Cycling '74, 554, 592
Cyclops, 554

Dada, 149–150
Davies, Char, 526
Davis, Joshua, 564–565
Deck, Barry, 112
Deleuze and Guattari, 564
Delvoye, Wim, 461
De Mol, Gerry, 275
Design By Numbers (DBN), xxiv, 552–523, 682
Designers Republic, The, 605
Dextro, 316
Dialtones (Levin et al.), 617–618
Digidesign, 587, 591
Dine, Jim, 606
DJ I, Robot Sound System, 506–509
Dodgeball, 617, 624
Domain Name System (DNS), 566
DrawBot, 169, 682, 684
Drawing with Computers (Wilson), 152, 217, 604
Drawn (Lieberman), 413
DuBois, R. Luke, 579
Duchamp, Marcel, 127, 633
Dunne, Tony, 634
Dürer, Albrecht 525, 612
DXF, 520, 529–531
Dynabook, 3

Eagle, 272
écal (école cantonale d'art de Lausanne), 271
Eclipse, 571, 625
ECMAScript, 681, 683
Edelweiss Series (Maywa Denki), 634
Edgerton, Harold, 295
Edison, Thomas, 579
Eighth Istanbul Biennial, 387
Eimart, Herbert, 580
Electronic Arts, 585
ELIZA, 101
Emacs, 516
Emigre, 605
End of Print, The (Blackwell), 605
Endless Forest, The (Tale of Tales), 274–277
Engelbart, Douglas, 205
Eno, Brian, 581
Enron, 268
Enzensberger, Hans Magnus, 564
EPS, 606
Euler's method, 7, 494
Every Icon (Simon), 565
Evolved Virtual Creatures (Sims), 295
Experiments in Art and Technology (E.A.T.), 633
Extend Script, 683
Eye magazine, 605
Eye Catching (Steinkamp), 386–389
EyesWeb, 554–555
EZIO (NIQ), 642

Feingold, Ken, 633
Ferro, Pablo, 327
Final Cut Pro (FCP), 383, 503
Final Scratch, 507
Fischinger, Oskar, 413
Fisher, Robert, 552
Flake, Gary William, 469
Flight404.com, 6
Flight Simulator, 525
Foldes, Peter, 315
FontLab, 170
Fontographer, 170
Fortran, 522
Fractal.Invaders (Tarbell), 156–159
Franceschini, Amy, 267
Franke, Uli, 260, 271
Free Radicals, 413
Friendster, 617
Fourier, Jean-Baptiste-Joseph, 584
Fourier transform, 585, 588, 590
Futurist, 279, 579

Gabo, Nam, 633
Galloway, Alexander R., 563
Game of Life, 461, 463, 465–466, 468, 475
Gardner, Martin, 461, 463
Garton, Brad, 581
Gerhardt, Joseph, 391–392
Gestalt psychology, 584

GIF, 95–96, 98–99, 421, 700–701
Girroir, Jonathan, 506–509
Google, 568, 617
GPS (Global positioning system), 619, 621
Graffiti, 223
GRASS, 681
Groeneveld, Dirk, 333
GNU Image Manipulation Program (GIMP), 95, 347, 355, 607–608
GNU Public License (GPL), 271
Gnutella, 566, 571
GPU (graphics processing unit), 536–537
Graphomat Z64 (Zuse), 603
Greenwold, Simon, 525
Greie, Antye (AGF), 503–504
Grzinic, Marina, 563
GUI (Graphical user interface), 435–436, 448, 450, 499, 604, 634, 679–680, 683, 685, 700
Gutenberg, Johannes, 111
Gutenberg archive, 433
Guttmann, Newmann, 580
Gysin, Andreas, 373

Hall, Grady, 379
Handel, George Frideric, 581
Hansen, Mark, 515–516, 634
Harmon, Leon, 604
Harvard University, xxi
Harvey, Auriea, 275
Hewlett-Packard (HP), 604, 610
Hawkinson, Tim, 633
Hawtin, Richie, 507
Hébert, Jean-Pierre, 217, 606
Hektor (Lehni, Franke), 260, 270–273
Henry, John, 507
Henry, Pierre, 580
Hiller, Lejaren , 581
Hoefler, Jonathan, 112
Hodgin, Robert, 6, 692
Hokusai, 612
Hongik University, 5
Hong, Leon, 5, 375
Hooke's law, 263, 487
Howard Wise gallery, 603
HTML (HyperText Markup Language), 9–11, 93, 268, 427, 549, 564–565, 568–569, 621, 624, 665–666, 684
HTTP (Hypertext Transfer Protocol), 567–569, 623
Huber, Daniel, 552
Huff, Kenneth A., 606
Hypermedia Image Processing Reference (HIPR), 552
HyperTalk, 522

IANA, 569
IBM, 315, 537, 580, 585, 604, 620, 702
IC (integrated circuit), 639, 647
I–Cube X (Infusion Systems), 642
IEEE 1394 camera, 556
If/Then (Feingold), 633
Igarashi, Takeo, 538
Igoe, Tom, 635, 648
Ikarus M, 170
Incredibles, The, 315
Internet Explorer, 565
Internet Protocol (IP), 566–567, 569, 589, 645
Impressionist, 279
Inaudible Cities: Part One (Semiconductor), 392
InDesign, 683
Infrared, 553, 621
Inge, Leif, 581
Inkscape, 77, 607–608
Installation (Greenwold), 526
Institute of Contemporary Arts (ICA), 101, 522
Intel Integrated Performance Primitives (IPP), 512, 555
Interaction Design Institute Ivrea (IDII), xxi, 634
i|o 360°, 565
I/O/D 4 ("The Webstalker"), 566
IRCAM, 554, 581, 592
Ishii, Hiroshi, 634
Ishizaki, Suguru, 327
ISO 216 standard, 611
Iwai, Toshio, 512, 549

James, Richard (Aphex Twin), 582
Jarman, Ruth, 391–392
Java, 7, 9–11, 146, 161–162, 263–264, 271, 499, 521–523, 528, 555, 564–565, 571, 574, 592, 622, 625– 626, 642, 663, 673, 677, 679–683, 686–690, 699–700
Java 2 Micro Edition (J2ME), 625
Java applet, 9–11, 264, 521, 656, 657, 675, 699
Java Archive (JAR), 10–11, 700
Java Core API, 271
JavaScript, 268, 271, 522, 624, 680, 681, 683
Java Virtual Machine (JVM), 680
Jeremijenko, Natalie, 548
Jevbratt, Lisa, 566
jMax, 592
Jodi, 563–566
Jones, Crispin, 634
Jones, Ronald, 275
Jonzun Crew, 508
JPEG, 95–96, 162, 421, 606, 611, 620, 701
JSyn (Java Synthesis), 592
Julesz, Bela, 603

Kay, Alan, 3
Kim, Tai-kyung, 5
Kimura, Mari, 582
King's Quest, 525
Klee, Paul, 217
Knowlton, Kenneth C., 315, 604
Krueger, Myron, 255, 512, 547
Kusaite, Lina, 275
Kuwakubo, Ryota, 634

La Barbara, Joan, 511
Langton, Chris, 469, 471
Putto8 2.2.2.2 (Rees), 524, 526
LaserWriter, 111, 604
Lee, Soo-jeong, 5
Led Zeppelin, 161
Legible City, The (Shaw, Groeneveld), 333
Lehni, Jürg, 260, 271–273
Leibniz, Gottfried Wilhelm, 61
Letterscapes (Cho), 327
LettError, 111, 168–170, 605
Levin, Golan, 259, 333, 511–512, 547, 617–618
Lewis, George, 582
LeWitt, Sol, 217
Li, Francis, 617
Lia, 316, 496
Lialina, Olia, 563–564
Licko, Zuzana, 112, 605
Lieberman, Zachary, 413, 512–512, 547
Lifestreams, 425–426
Limewire, 571
Lingo, 522–523, 555, 565, 683, 686–687, 689, 691
Linklater, Richard, 383
Linotype, 111
Linux, 4, 9–11, 508, 521, 568–569, 625, 645, 649

Listening Post (Rubin, Hansen), 514–517
LISP, 101
LiveScript, 683
Local area network (LAN), 568–569
Logo, xxiii, 2, 217, 522, 681
Lovink, Geert, 564
Lozano-Hemmer, Rafael, 546, 548
Lucent Technologies, 515
Lucier, Alvin, 590
Luening, Otto, 580
Lüsebrink, Dirk, 549
Lye, Len, 413

Machine Art exhibition, 291, 633
Machine Perception Laboratories, 549
MacMurtrie, Chico, 549
Macromedia Director, 166, 387–388, 554–555, 642, 683, 686
Maeda, John, xix, xxiii, xxiv, 3, 5, 158, 333, 564, 606, 682
Malka, Ariel, 372
Makela, P. Scott, 605
Mandelbrot, Benoit, 153
Manovich, Lev, 565
Marble Madness, 525
Marconi, Guglielmo, 579
Marey, Étienne-Jules, 295
Mark of the Unicorn Digital Performer, 591
Markov chain, 581
Marx, Karl, 267–268
Massachusetts Institute of Technology (MIT), xix, xxiii, xxiv, 327, 634, 680, 682, 693, 695
Masterman, Margaret, 101
Mathews, Max, 580, 586, 591, 683
MATLAB, 522
Max/MSP/Jitter, 2, 503–504, 515–517, 522, 554–555, 571, 580, 592, 642, 683–685
Maya Embedded Language (MEL), 680, 683
Maywa Denki, 634
McCarthy, John, 101
McCartney, James, 592
McCay, Winsor, 315
McLaren, Norman, 413
Medusa, 387
MEL, 680, 683
Mendel, Lucy, 507
Messa di Voce (Tmema et al.), 510–513, 547
Metrowerks Codewarrior, 512
Microsoft, 4, 111, 169, 436, 508, 525, 537, 585, 702
Microsoft Direct3D, 537
Microsoft Visual Basic, 436
Microsoft Windows, 9, 11, 264, 367, 421, 435–436, 511, 521, 568, 625, 645, 649, 665–666, 685
MIDI (Musical Instrument Digital Interface) 162, 554, 588–589, 591–592, 618, 621, 623, 642, 645, 683, 685
Mignonneau, Laurent, 549
MIME, 623
Mims, Forest M., III, 648
Mini-Epoch Series, The (Semiconductor), 390–393
Mini Movies (AGF+SUE.C), 500, 502–505
Minitasking (Schoenerwissen/OfCD), 562, 566
Minsky, Marvin, 547
MIT Media Laboratory, xxiii, 327, 634, 680, 682, 702
MixViews, 591
MP3, 162, 421, 585, 621, 623
MPEG–7, 549
Mobile Processing, 521, 622–626, 683
Mohr, Manfred, 217, 602, 606
Möller, Christian, 549
Moore, F. Richard, 592
Mophun, 625
Morisawa, 605
Motion Theory, 378–381
MTV, 384
[murmur], 618
Museum of Modern Art, The (MOMA), 291, 633
MUSIC, 580, 591
Musique concrète, 580–581
Muybridge, Eadweard, 295, 373
Myron, 555
MySQL, 267–268
Myst, 525

Nakamura, Yugo, 565
Nake, Frieder, 217, 603
Napier, Mark, 566
Napster, 507, 571
Nees, Georg, 217, 603
Nelson, Ted, 3
"net.art", 563–564
net.art (Baumgärtel), 564
net.art 2.0 (Baumgärtel), 564
NetBeans, 625
Netscape Navigator, 565, 683
Newton, Isaac, 477, 488
New York University (NYU), 6, 634
New York Times, The, 150
Ngan, William, 497
Nimoy, Josh, 512
Noll, A. Michael , 217, 603
Nokia, 517, 618–619, 625
Nmap, 569
NSA (National Security Agency), 268
NTNTNT (Cal Arts), 564
NTSC, 367
NTT DoCoMo's i–Mode, 624
Nuendo, Steinberg, 591
null, 40, 701
NURBS (Non-uniform Rational B-splines), 526
nVidia, 537
Nyquist theorem, 585

OBJ, 529–531
Objectivity Engine, The (Paterson), 164–167
Oliveros, Pauline, 582
Olsson, Krister, 589
Once-Upon-A-Forest (Davis), 564
On, Josh, 267–268
oN-Line System (NLS), 205
OpenCV, 512, 555
OpenGL, 512, 520, 528, 531, 537, 554, 684
Open source, 4, 268, 271, 512, 521, 555, 591, 625–626, 640, 684
OpenType, 111, 169
Oracle database, 264
OSC (Open Sound Control), 516–517, 571, 589
oscP5 (Schlegel), 571
Osmose (Davies), 526
O'Sullivan, Dan, 635, 648
Oswald, John, 581
Owens, Matt, 565

Pad, 435
Paik, Nam June, 633
PAL, 367
Palm Pilot, 223, 625
Palm OS, 625
Panasonic, 625
Papert, Seymour, 2, 217
Parallax, 640
Parallel Development, 516
Pascal, 522

Paterson, James, 165–166, 316, 565, 606
Paul, Les, 580
PBASIC, 642, 681
PC, 10, 227, 388, 625, 665, 682
PCB (printed circuit board), 639, 640
PCM (pulse–code modulation), 585–586, 699, 702
PDF, 520, 606–608, 682
Pelletier, Jean-Marc, 554
Penny, Simon, 549
Perl, 146, 515–517, 522–523, 565, 571, 681, 684
Perlin, Ken, 130
Personal area network (PAN), 621–622
Petzold, Charles, 648
Phidgets, 642
Philips, 634
PHP, 267–268, 522–523, 565, 682, 684
PHPMyAdmin, 268
Physical Computing (O'Sullivan, Igoe), 648
Piano Phases (Reich), 293
PIC (Microchip), 272, 640
PIC Assembler, 271–272
PIC BASIC, 681
Pickard, Galen, 507
Pickering, Will, 516
Pixar, 315
Pixillation (Schwartz), 315
PNG (Portable Network Graphics), 95–96, 98–99, 606, 622, 701
Pocket PC, 625
PoemPoints, 617
Pong, 256, 590, 618
PortAudio, 512
PostScript, 111, 143, 169–170, 522, 604–605, 681
Poynor, Rick, 605
Practical Electronics for Inventors (Scherz), 648
Practice of Programming,The (Kernighan, Pike), 252
Praystation (Davis), 564
Public Enemy, 581
Puckette, Miller, 2, 592, 684
Pulse–code modulation (PCM), 585–586, 699, 702
Pure Data (Pd), 592, 684–685
Python, 146, 170, 517, 522–523, 681–682, 684

Q*bert, 525
Quartz Composer, 684
Qualcomm, 625
Quest3D, 275–276

R, 515, 517
Raby, Fiona, 634
Radial, 503–504
RAM, 701
RandomFont Beowolf (LettError), 111, 168–170, 605
Rauschenberg, Robert, 606
Ray Gun, 605
Razorfish, 565
RCA Mark II Sound Synthesizer, 580
Readme!, 563
Real-Time Cmix, 592
Rees, Michael, 526
Reeves, Alec 585
Reich, Steve, 293
Reichardt, Jasia, 522
Reiniger, Lotte, 315
RenderMan, 315
R.E.M. "Animal" (Motion Theory), 378–381
ResEdit, 170
Resnick, Mitchel, 471, 680
Reynolds, Craig, 295, 473, 497
Rhino, 271, 537
Rich, Kate, 548
Riley, Bridget, 151
Ringtail Studios, 275
Risset, Jean-Claude, 581
RoboFog, 170
Rokeby, David, 548, 554
Rotoshop, 383–384, 413
Royal Academy of Arts, 169
Royal College of Art, 634
Rozin, Danny, 549
RS-232, 639, 554, 640, 645
Rubin, Ben, 515, 634
Ruby, 681, 684
Ruby on Rails, 684
Runge-Kutta method 7, 494
Russolo, Luigi, 579

Sabiston, Bob, 383–384, 413
Saito, Tatsuya, 198, 529, 568
Samyn, Michaël, 275
Sauter, Joachim, 549
Schaeffer, Pierre, 580
Scheme, 522
Scherz, Paul, 648
Schiele, Egon, 217
Schlegel, Andreas, 498, 571
Schmidt, Karsten (a k a toxi), 4, 518
Schoenerwissen/OfCD, 562
Schöffer, Nicolas, 633
Schumacher, Michael, 582
Schwartz, Lillian, 315
Scientific American, 461, 463
Scratch, 680
Screen Series (Snibbe), 549
Scriptographer (Lehni, Franke), 270–273, 683
Seawright, James, 633
sed, 684
Semiconductor, 390–393, 646
Sessions, Roger 580
Sester, Marie, 549
Shannon, Claude, 669
Shape of Song (Wattenberg), 160–163
Shaw, Jeffrey, 333
Shiffman, Daniel, 6
Shockwave Flash (SWF), 158, 565
Short Messaging Service (SMS), 617, 619, 621
SHRDLU, 101
sh/tcsh, 515, 684
Shulgin, Alexi, 563–564
Silicon Graphics, 529, 537
Simon, John F. Jr., 413, 565
SimpleTEXT, 618
Sims, Karl, 295
Sinclair Spectrum, 264
Singer, Eric, 554
Sketchpad, 217
SketchUp, 538
Slacker, 383
Slimbach, Robert, 112
Smalltalk, 685
Smith, Laura, 275
Snake, 618
Snibbe, Scott, 413, 549
Social Mobiles (SoMo), 634
Sodaconstructor (Burton), 262–265, 413, 499
Soda Creative Ltd., 263–264
SoftVNS, 554
Solidworks, 537
Sommerer, Christa, 549
Sonami, Laetitia, 582
Sonic Inc., 392
Sony, 634
Sony Ericsson, 625
Sorenson, 388
Sorting Daemon (Rokeby), 548, 554
Sound Films, 392

Spark Fun Electronics, 640
SQL (Structured Query Language), 685
Srivastava, Muskan, 5
Standards and Double Standards (Lozano-Hemmer), 547–548
Star Wars, 315
Strausfeld, Lisa, 327
Stedelijk Museum, 218
Stehura, John, 315
STEIM (Studio for Electro-Instrumental Music), 554
Steinkamp, Jennifer, 387–388
Stipe, Michael, 379–380
Stockhausen, Karlheinz, 580
Stone, Carl, 582
Stop Motion Studies (Crawford), 316
Studies in Perception I, (Knowlton, Harmon), 604
Substrate (Tarbell), 6, 154, 156–159
Sudol, Jeremi, 507
Suicide Box, 548, 554
Sun Java Wireless Toolkit, 625
Sun Microsystems, 521, 537, 625, 682
SuperCollider, 571, 592, 685
Sutherland, Ivan, 217
SVG (Scalable Vector Graphics), 77, 520, 606, 624
Symbian, 625
Synergenix, 625

Tale of Tales, 274–277
Talmud Project (Small), 327
Takeluma (Cho), 327
Takis, 633
Tarbell, Jared, 6, 155–156, 606
Tangible Media Group (TMG), 634
TARGA, 368, 606, 702
Tate Gallery, 218
T|C Electronics Powercore, 587
tcpdump, 568–569
TCP/IP, 554, 569, 589
Technics, 507
Teddy (Igarashi), 538
Telephony (Thompson, Craighead), 618
TeleNav, 619
Teleo (Making Things), 642
Tesla, Nikola, 579
Text-to-speech (TTS), 516–517
They Rule (On et al.), 266–269
Thomson, Jon, 618
TIFF, 368, 507, 606, 608, 611, 702
Toy Story, 315
Tmema, 510–513
Transmission Control Protocol (TCP), 569
Tron, 315
Truax, Barry, 581
TrueType, 111
Tsai, Wen–Ying, 633
TurboGears, 684
Turkle, Sherry, 5
Turux (Lia, Dextro), 316
Type, Tap, Write (Maeda), 333
Tzara, Tristan, 150

Überorgan (Hawkinson), 633
Unicode, 432, 665–668
University of California
  Berkeley, 589
  Los Angeles (UCLA), xxi, 4, 5, 574
  San Diego (UCSD), 549
University of Cincinnati (UC), xxiii
University of Genoa, 554
UNIX, 227, 435, 517, 569, 645
U.S. Army Ballistic Missile Research Laboratories, 603
USB, 556, 640–645, 701
User Datagram Protocol (UDP), 554, 569, 589
Ussachevsky, Vladimir, 580
UTF-8, 665
Utterback, Camille, 549

Valicenti, Rick, 605
van Blokland, Erik, 169–170
van Blokland, Petr, 170
VanDerBeek, Stan, 315
Vanderlans, Rudy, 605
van Rossum, Just, 169
Vaucanson's Duck, 461
Vehicles: Experiments in Synthetic Psychology (Braitenberg), 473
Venice Biennale, 391
Verschoren, Jan, 275
"Video Games and Computer Holding Power" (Turkle), 5
Videoplace (Krueger), 547
Visual Language Workshop (VLW), 327, 702
Visual programming languages (VPL or VL) , 679–680
Vitiello, Stephen, 582
VLW font format, 112, 702
Vogel, Peter, 632, 633
Von Ehr, Jim, 170
Vonnegut, Kurt, 507
von Neumann, John, 461
Vorbis codec, 585
Voxel, 527
vvvv, 685

Wacom, 383
Waking Life, 382–385, 413
Walt Disney, 315, 379
Wang, Ge, 592
Warner Bros. Records, 379
Wattenberg, Martin, 161–162, 606
Watz, Marius, 374
WAV, 585–586, 621, 623, 702
Wayfinder Systems, 619
Wegman, William 606
Weizenbaum, Joseph, 101
Whitney, James, 315
Whitney, John, 315
Whitney Museum of American Art, 516
Wilhelm Imaging Research, 610
Wilson, Mark, 152, 217, 604
Winograd, Terry, 101
Wiring, 521, 633, 640, 641, 645–646, 648–649, 685
Wright, Frank Lloyd, 333
Wrongbrowser (Jodi), 566
Wolfram, Steven, 461, 463–464, 467, 475
Wolfenstein 3D, 525
Wong, Andy, 507
Wong, Yin Yin, 327

Xenakis, Iannis, 581
Xerox Palo Alto Research Center (PARC), 3, 205
Xerox, 507
xHTML Mobile, 624
XML, 421, 427–428, 520, 549, 621, 624, 702

Yamaha Digital Mixing Engine (DME), 516
Yellow Arrow, 618
Youngblood, Gene, 388

Ziggurat (font), 112
Zooming user interface (ZUI), 435